Architecture et protocoles applicatifs pour la chorégraphie de services dans l'Internet des objets

Sylvain Cherrier

UNIVERSITÉ PARIS-EST

ÉCOLE DOCTORALE MSTIC

Mathématiques et Sciences et Technologie de l'Information et de la Communication

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université Paris Est

Mention : Informatique

Présentée et soutenue par

Sylvain Cherrier

Architecture et protocoles applicatifs pour la chorégraphie de services dans l'Internet des objets

Thèse dirigée par Gilles Roussel

préparée au LIGM

soutenue le 25 novembre 2013

Jury :

Rapporteurs :	Ahmed Karmouch	-	University of Ottawa (Canada)
	Dominique Gaïti	-	Université de Technologie de Troyes
Directeur :	Gilles Roussel	-	Université Paris-Est
Examinateurs :	Olivier Festor	-	Université de Lorraine (LORIA)
	Jérémie Leguay	-	Thales Communications & Security
	Cyril Nicaud	-	Université Paris-Est
	Jean-Marie Bonnin	-	Télécom Bretagne
Co-Encadrants :	Yacine Ghamri-Doudane	-	Université Paris-Est
	Stéphane Lohier	-	Université Paris-Est

Remerciements

Entreprendre une thèse en parallèle d'une activité professionnelle d'enseignant agrégé occasionne quelques difficultés, et nécessite le soutien de l'entourage, ce qui explique la longueur de ces remerciements.

Je tiens tout d'abord à remercier les membres de ce jury : Les professeurs Ahmed Karmouch de l'Université d'Ottawa et Dominique Gaïti de l'Université de Technologie de Troyes qui ont accepté d'être les rapporteurs de cette thèse, ainsi que leurs corrections qui ont permis la finalisation de ce manuscrit. Je remercie aussi le Dr Olivier Festor qui dirige l'équipe de recherche MADYNES du LORIA de l'Université de Lorraine et le Dr Jérémie Leguay du groupe Thales (Communications & Security) d'avoir eu l'amabilité d'accepter mon invitation à ce jury.

Au sein de notre école doctorale, je voudrais exprimer ma reconnaissance à mon directeur de thèse le professeur Gilles Roussel pour son soutien et avec lequel les échanges sont si simples et directs. Je n'oublie pas mes encadrants de thèse, Yacine Ghamri-Doudane et Stéphane Lohier pour m'avoir entraîné dans cette aventure, écouté, contredit, conseillé, relu, encouragé, écouté, contredit, conseillé, relu, encouragé, etc.

Je voudrais aussi remercier l'Institut Gaspard Monge et le laboratoire associé. À la fois leurs directeurs, respectivement Serge Midonnet et Marie-Pierre Béal, qui ont autorisé le déroulement de cette thèse et accordé un aménagement de mon temps de travail sur les 3 dernières années, mais aussi l'ensemble des collègues, chercheurs, personnels administratif, enseignants, tous ont été tour à tour sources d'idées, interlocuteurs et soutiens. Cyril, Rémy, Dominique, Marc, Pascale, Corinne, Angélique, Line, Claire, Florence et Philippe, et les autres personnels du LIGM. Les échanges avec vous, que ce soit sur des points techniques, sur des pistes à explorer, ou tout simplement d'un point de vue humain m'ont beaucoup apporté. À tous merci.

Je n'oublie pas les membres du groupe PASNet, ceux que j'ai déjà remercié plus haut, les autres membres du groupe Akim, Abderrezak, et les thésards du bout du couloir, au 4ième, et plus particulièrement Ibtissem, Nadia, ainsi qu'Erwan et Philippe, les ingénieurs chargés de mettre en musique nos idées les plus étranges. Une mention particulière pour Ismail Salhi, infatigable moteur, perpétuel fédérateur, inventeur, artiste et créateur, toujours à l'écoute et toujours en avance, tu vas me faire regretter que cette thèse soit terminée... Une mention spéciale pour mes relecteurs bénévoles : Merci Etienne, je ne sais pas comment tu as trouvé le temps dans ton planning. Et bien sûr Aurélie et Florence, véritables petits diables de la grammaire, de la conjugaison, du mot juste et de l'emploi fidèle. Merci pour tout cela.

Reste enfin ma famille à remercier, à mes parents qui ont eu la gentillesse d'attendre 20 ans que je daigne enfin travailler à l'école, à mes enfants celle de me laisser faire mes devoirs, mes sœurs pour leur indéfectible soutien. Et enfin à celle qui se reconnaîtra, merci de m'avoir laissé tout ce temps et d'avoir supporté mes doutes...

Résumé : Les défis que l'Internet des objets posent sont à la mesure des transformations que cette technologie est susceptible d'entraîner dans notre rapport quotidien à notre environnement. Nos propres objets, et des milliards d'autres, disposeront de capacités de traitement des données et de connexion au réseau, certes limitées mais effectives. Alors, ces objets se doteront d'une dimension numérique, et deviendront accessibles d'une façon tout à fait nouvelle. Ce n'est pas seulement la promesse d'un accès original à l'objet, mais bel et bien l'avènement d'une nouvelle perception et interaction avec ce qui nous entoure. Les applications de l'Informatique *ubiquitaire* utiliseront majoritairement les interactions entre objets, et la somme de leurs actions/réactions offrira une véritable valeur ajoutée.

Mais l'hétérogénéité des composants matériels et des réseaux empruntés freine considérablement l'essor de l'Internet des objets. L'objectif de cette thèse est de proposer une solution effective et le cadre nécessaire à la construction de telles applications. Nos travaux s'inscrivent parmi les propositions de frameworks, dans lequel chaque objet, autonome, est reconfigurable à distance et fournit de services. Après avoir montré la pertinence des solutions chorégraphiées et quantifié le gain acquis sur des structures de communication arborescentes, nous présenterons D-LITe, notre framework de logique distribuée. Grâce à son approche REST assurant l'interopérabilité dans l'assortiment des composants et réseaux de l'Internet des objets, le framework D-LITe, hébergé par chaque objet (et adapté à ses contraintes), en propose un contrôle à distance, aussi bien pour sa reprogrammation dynamique que pour les échanges avec ses partenaires. Nous poursuivrons en présentant SALT, le langage de programmation destiné à D-LITe, basé sur les transducteurs à états finis. Outre son expressivité étendue aux particularités du domaine, SALT accorde un accès aux fonctionnalités de l'objet, au travers d'une couche d'abstraction matérielle. Enfin, profitant de la standardisation offerte par D-LITe pour la programmation de chaque composant en particulier, une solution de composition, BeC^3, va offrir un moyen efficace pour construire une application complète par assemblage des comportements distribués, tout en respectant la cohérence des interactions entre objets, par l'intermédiaire d'une abstraction des échanges et de leur modélisation. Aussi sommes-nous, par la résolution des problématiques rencontrées à chacun des différents niveaux, capables de présenter une solution simple, cohérente et fonctionnelle à même de bâtir réellement et efficacement, des applications robustes pour l'Internet des objets.

Mots clés : Internet des objets ; Chorégraphie de services ; Composition de services

Table des matières

Chapitre 1

Introduction

Objets inanimés, avez-vous donc une âme ?

Milly ou la terre natale
A. de Lamartine

1.1 Présentation de la thèse

Des objets et des utilisateurs qui interagissent, des bâtiments qui guident leurs visiteurs et les renseignent, configurent leurs appareils électroniques portatifs, leur proposent des services contextualisés ou encore des villes intelligentes gérant de façon autonome leurs ressources, sont quelques unes des promesses de l'Internet des objets. Cette extension du réel par l'ajout de fonctionnalités de communication et de traitement donne à nos objets de tous les jours de nouvelles possibilités d'interactions. Notre quotidien s'enrichit alors d'une dimension étendue et virtuelle. L'extension de l'Internet, irriguant de plus en plus finement et profondément notre espace, à l'image de vaisseaux sanguins ou de racines qui se ramifient et s'étendent (on parle alors d'*Internet Capillaire*), complète la perception du monde qui nous entoure d'un accès virtualisé.

Mais cette interaction n'est pas limitée à la relation homme/objets. En effet, pour profiter de la plus-value apportée par l'ajout des capacités de communication et de traitement aux éléments matériels du monde réel, nos objets sont amenés à interagir entre eux ou avec des outils virtuels, tels des sites web, des services réseaux, des outils de localisation, etc. Les besoins en matière de communications *Machine à Machine* (M2M) sont conséquents. M. Uusitalo [128] estime à 7 mille milliards le nombre d'objets reliés au service de 7 milliards d'humain en 2017, D. Evans [60] l'évalue quant à lui à 50 milliards en 2020, et le rapport ITU Internet of Things [121] prévoit que, dès 2020, le nombre d'échanges entre machines (T2T, *Thing to Thing*) dépassera celui entre humains et machines (*H2T, human to Thing*). Ces multitudes d'objets connectés permettent d'envisager un large éventail de services automatisés, intelligents, capables de réactions, d'adaptation, dans un milieu stabilisé tel qu'un lieu d'habitation, une usine, un entrepôt, une centrale ou, à plus grande échelle, une ville intelligente (*Smart cities* [76]). Ils sont aussi adaptés aux situations critiques (accidents, catastrophes), dans lesquelles il est impératif de reconstruire le plus rapidement possible les systèmes de communication primordiaux, les contrôles automatisés indispensables, ou encore d'implémenter en urgence des dispositifs de

recherche, d'assistance, alors que l'infrastructure sous-jacente est partiellement, voire totalement inopérante.

Cependant, l'émergence des matériels, outils et concepts liés à ce domaine prometteur se heurte à la difficulté d'appréhender les enjeux, méthodes et impasses que toute évolution technologique engendre. Dans cette absence de visibilité réside l'un des principaux freins au développement de l'Internet des objets, tant dans le choix des méthodes de transmission des informations entre objets (requêtes, centralisation des données, traitement distribué, data-centric, event-driven) qu'au niveau des protocoles régissant les échanges (SOAP, HTTP, autre ?) ou encore, plus généralement, sur les usages de ces outils. Confronté à une tour de Babel technologique, chacun (l'industriel comme l'académique, ou l'utilisateur) hésite à s'engager, ou doit, s'il le fait, avoir soit recours à une large palette de connaissances, d'outils, et de modes opératoires très variés et spécifiques, soit volontairement ignorer tout un pan du domaine, risquant alors de se couper des apports et avancées proposés par d'autres acteurs. L'incompatibilité des différentes strates proposées pénalise la réalisation de solutions élaborées et compromet leur universalité, et donc leur succès.

L'étude de la standardisation des technologies, sa nécessité même, garante d'un socle du développement d'une activité industrielle et de l'interopérabilité des solutions des différents intervenants, fait partie des missions de la recherche concernant le développement actuel en informatique. L'Internet des objets s'appuie sur un vaste éventail de matériels, d'infrastructures réseaux et de protocoles de communication et d'application. Les caractéristiques de chacun de ces élément génèrent des avantages et inconvénients particuliers. Ainsi, les solutions du coté Internet des PC (serveurs, services, protocoles réseaux, infrastructure réseau, clients) sont-elles bien connues, maîtrisées et standardisées. L'Internet des objets doit permettre l'accès du réseau des réseaux à de nouvelles architectures, les réseaux de capteurs notamment. Bien que l'Internet des objets ne se limite pas aux *Wireless Sensors and Actuators Networks* (WSAN), l'intégration des WSAN influe grandement sur l'usage de l'ensemble, car les caractéristiques des chaînes sont souvent limitées à celles des maillons les plus contraints. L'Internet ne pourra bénéficier des apports des WSAN, et ainsi redessiner le contour de sa zone d'influence qu'en intégrant et respectant les limitations des capteurs et effecteurs du WSAN. Que ce soit par l'usage de passerelles ou par une adaptation des protocoles utilisés, les efforts de la recherche doivent se concentrer sur les différentes possibilités d'établir la jonction entre ces deux mondes, et leurs conséquences.

C'est dans ce contexte que cette thèse s'inscrit puisqu'elle s'organise autour de la proposition de solutions qui découlent du constat des difficultés à réellement réaliser une intégration des technologies propice à l'éclosion d'architectures efficaces et novatrices pour l'Internet des objets. L'émergence de normalisation et de standardisation à tous niveaux offrira une stabilité propre à rassurer les différents acteurs du domaine, ouvrant la voie à des développements plus sûrs et pérennes, facilitant les collaborations, et proposant des solutions pour chacune des strates évoquées. L'extension de l'Internet à ces nouveaux réseaux d'objets contraints soulève la question de la pertinence et/ou de l'adaptation des protocoles applicatifs à l'œuvre, des ap-

proches de programmation et des architectures logicielles, capables d'à la fois offrir la continuité promise dans l'appellation "Internet" tout en respectant les spécificités de chacune des parties prenantes. Les applications proposées devront simultanément se fondre dans la palette des outils usuels que l'internaute manipule, et intégrer ces objets intelligents dans leurs interactions.

L'objectif n'est pas tant d'explorer des utilisations innovantes que de fournir le cadre qui facilitera leur apparition. Le foisonnement des technologies et des architectures génèrent une importante complexité dès qu'il s'agit d'interconnecter les différents mondes qui constituent l'Internet des objets. Le but recherché est de concevoir des mécanismes capables de nous affranchir d'éventuelles adaptations, réalisées au cas par cas et par conséquent potentiellement pénalisantes en termes de rapidité, d'efficacité, et d'évolutivité, pour bâtir une proposition globale et cohérente de bout en bout, dans laquelle chacun pourra s'immiscer d'autant plus facilement que les contraintes seront légères et la portée de la proposition sera générique.

1.2 Contributions

L'Internet des objets fédère une large gamme de matériels, technologies et protocoles avec l'ambition d'offrir des interactions homme/machine ou machine/machine améliorées, plus automatiques et autonomes. Puisque les appareils qui nous entourent sont dotés de capacités de traitement de données et de connectivité, la volonté d'offrir une solution d'informatique pervasive [1] promeut leur nécessaire interconnexion au réseau standard, banalisé et extrêmement répandu qu'est l'Internet. Et parmi l'ensemble des protocoles applicatifs supportés par cette technologie, le Web est l'un des plus communément admis, répandu, non filtré et que les multiples usages ont validé. Le *Web des objets* (WoT) en est le prolongement, alliant l'universalité et la simplicité du protocole applicatif HTTP à l'utilisabilité des objets qui nous entourent (les nôtres, et ceux publics et accessibles) et dont l'alliance nous autorisera, à terme, à capter et interagir de plus en plus sur notre environnement. Dans cette thèse, nous avons choisi de nous concentrer sur l'approche Web, particulièrement adaptée aux échanges, même si, le plus souvent, nous employons le terme générique d'*Internet des objets* (IoT).

Cette thèse décrit une solution globale et fonctionnelle, bâtie autour de trois contributions principales, et qui sont décrites dans les sous-sections qui suivent.

1.2.1 Vers une organisation distribuée

Résoudre le besoin d'applications de l'Internet des objets peut s'envisager sous l'angle de l'*architecture orientée service* (SOA) comme le montre l'orientation actuelle des travaux de recherche. L'organisation de l'application en tant que collaboration de services peut s'appréhender selon deux modes : un mode centralisé et un

1. Néologisme signifiant "*diffus et omniprésent*".

mode distribué. Laquelle de ses deux architectures logicielles est la plus efficace au vu des contraintes de l'IoT ?

Ces deux organisations se retrouvent à la fois dans l'Internet des PC et dans les WSAN : ces derniers sont *automatisés*, ou *semi-automatisés*, c'est-à-dire que les actions des effecteurs sont soit directement pilotées par les capteurs, soit sous le contrôle d'un élément central, hors WSAN, qui déduit des données captées et remontées les actions à entreprendre. Notre travail se concentre tout d'abord sur l'évaluation des impacts de ces deux organisations dans une infrastructure générique d'un réseau de capteurs, présentée sous forme d'arbre. Les gains offerts par la chorégraphie de services (c'est-à-dire selon une approche distribuée), comparée à une orchestration de services, sont ainsi quantifiés sur des arbres remarquables ou communs, révélant alors les incidences de l'organisation applicative sur la longueur du chemin parcouru par les messages échangés. Des expériences utilisant des capteurs/effecteurs dans un réseau réel permettent ensuite de valider les apports de la chorégraphie par rapport à l'orchestration, en donnant des valeurs effectives comprises entre les bornes prédites par notre étude. Aussi les organisations distribuées contribuent-elles mieux au respect des contraintes d'énergie des WSAN, tout en s'adaptant à la nature particulière des applications de l'Internet des objets. Ainsi, cette première étude et les conclusions afférentes représentent la première contribution de cette thèse.

1.2.2 Infrastructure de machines virtuelles pour application distribuée

Si les organisations chorégraphiées sont préférables, les applications de l'Internet des objets se bâtissent cependant en combinant une large gamme d'éléments aux caractéristiques si dissemblables qu'il devient alors difficile de réaliser les programmes décrivant le rôle de chacun. La grande diversité de matériel, langage, réseau et puissance de calcul, ainsi que l'absence de points communs entre toutes ces parties prenantes pénalisent le développement d'applications distribuées dont l'analyse a pourtant montré la pertinence dans le cadre de l'IoT. Ce constat a donné lieu à deux contributions.

Nous proposons en premier lieu une plateforme logicielle, D-LITe, installée sur tout objet impliqué, uniformisant sa représentation et standardisant sa configuration à distance. Afin de résoudre l'hétérogénéité des matériels qui s'agrègent dans l'Internet des objets, D-LITe en fédère l'ensemble en proposant une machine virtuelle chargée de représenter, de façon universelle, chaque objet. Aussi s'appréhendent-ils tous de la même façon du point de vue du programmeur et de l'utilisateur. Les spécificités de chacun des objets sont résolues par l'entremise d'une couche d'abstraction matérielle, chargée de cacher sur un vocabulaire normalisé la diversité des matériels réellement à l'œuvre. L'expression du service attendu, au travers de notre langage SALT, utilise des transducteurs à états finis, pour leur universalité, et leur facilité à être interprétés sur tout type d'objets. Des extensions pour les alphabets d'entrée et de sortie dotent notre langage SALT de l'expressivité nécessaire à la réalisation

d'algorithmes plus riches. L'introduction de prédicats dans les transducteurs donne à nos services la capacité de gérer des variables et de se combiner de façon plus efficace, sans altérer la mécanique propre aux transducteurs.

Le problème de la configuration à distance, rendue nécessaire par l'intégration des capteurs et effecteurs dans cet Internet étendu aux objets, est résolu grâce à l'utilisation des protocoles standards et de l'approche REST de D-LITe. Cette démarche consistant à appréhender l'objet comme un fournisseur de services nous permet la proposition d'une inversion des démarches traditionnelles de composition, particulièrement propice à répondre aux besoins de l'IoT. Plutôt que d'assembler des services pré-existants, notre approche consiste à déployer à la volée le service nécessaire pour résoudre les besoins de l'utilisateur. La démarche top-down de D-LITe (l'expression des besoins de l'utilisateur déclenchent le déploiement de la solution correspondante) montre sa pertinence dans cet environnement particulier, fait d'une multitude d'îlots de raisonnement, collaborant pour un objectif commun.

Enfin, puisque les applications distribuées sont particulièrement sensibles aux pertes de messages, et que la participation des WSAN est susceptible d'en générer, une seconde étude nous a permis de nous intéresser à des dispositifs capables d'améliorer la fiabilité des applications Internet des objets d'une manière générale. Nous proposons, dans SALT, un mécanisme chargé d'organiser des resynchronisations entre les participants pour compenser les éventuels décalages qui pourraient survenir entre nœuds. Ce dispositif, reprenant lui aussi l'approche distribuée de D-LITe afin de rester cohérent avec l'ensemble de la proposition, permet au concepteur de l'application la construction d'une sur-couche d'arborescences de points de contrôle, sur lesquels il est possible de déclencher des cascades de vérification. Aussi, tout décalage par rapport à des équilibres prédéfinis peut être compensé. L'étude se clôt sur un comparatif de l'efficacité de la solution, capable de contenir une éventuelle dérive de l'application dans un intervalle prédéfini, évaluée par rapport au surcoût en termes d'échanges au sein de la structure.

1.2.3 Abstraction des échanges et compositions agiles

La composition chorégraphiée de services, exécutés sur un ensemble de machines virtuelles, uniformisant les matériels participant à l'Internet des objets, fournit le cadre nécessaire à la création d'applications, plus génériques, et portables. Cependant, leur réalisation, basée sur une organisation des échanges de messages entre objets, demande au programmeur de la rigueur et une connaissance précise du détail des alphabets entrant et sortant de chaque transducteur à l'œuvre. Afin d'alléger ces contraintes, et au regard de la relative simplicité des codes déployés sur chaque objet, nous avons imaginé BeC^3, une solution de composition agile des parcelles de codes distribués. Grâce à la caractérisation des échanges et à la modélisation des dépendances entre interlocuteurs, nous pouvons proposer une solution de *mashup* [2] de codes génériques, réalisés et partagés entre utilisateurs. Chaque code générique

2. Outil qui permet de connecter des ressources entre elles.

est référencé dans le système de gestion de BeC^3, et dispose d'une structure d'informations qui lui sont associées. Outre son identifiant, son code SALT à destination de la machine virtuelle de D-LITe, les fonctionnalités requises sur l'objet (selon la codification établie offerte par la couche d'abstraction matérielle), BeC^3 stocke, pour chacun d'eux, ses contraintes en termes d'échanges recensés dans le cadre du modèle que nous fournissons. L'étude des relations entre les différentes exigences et offres de chaque code déployé sur chacun des objet permet la construction d'ensembles et multi-ensembles descriptifs des échanges possibles. BeC^3 emprunte à la théorie des ensembles pour édicter et analyser les relations attendues entre ces différents ensembles, représentant les interactions globales de l'application à projeter sur les nœuds. Il contient un moteur de composition chargé de vérifier la conformité de la composition souhaitée, au regard du modèle d'interactions, avant le déploiement réel des codes sur les machines virtuelles présentées au chapitre précédent. L'abstraction des échanges proposée facilite l'utilisation et le partage de codes génériques, développés par une communauté, dans une approche participative (*Crowd*), dans l'objectif de disposer d'un panorama riche et diversifié de comportements d'objets, en constante évolution, pour la conception de compositions de plus en plus diverses et complexes.

1.3 Organisation du document

Les chapitres de cette thèse s'organisent selon une structure logique en débutant par une étude des impacts sur les réseaux WSAN des différentes organisations des architectures de services, pour aboutir à notre vision de génération assistée d'applications Internet des objets, en passant par la mise en place des protocoles et des couches d'abstraction pour une interopérabilité des objets entre eux.

Plus précisément, la trame de ce manuscrit est construite en respectant la progression suivante :

Nous commencerons, dans le chapitre 2, par un état de l'art de l'Internet des objets et des différents domaines qui y fusionnent, au niveau tant des différents réseaux mis en œuvre (et notamment les réseaux de capteurs) que des protocoles applicatifs et des architectures logicielles, au travers des approches, problématiques et solutions proposées par le monde académique.

Le 3^{e} chapitre se concentre sur la première contribution de cette thèse, à savoir la quantification des gains, en termes de longueur de chemin parcouru par des messages dans un arbre, selon qu'une application s'organise suivant un mode distribué ou centralisé. Les résultats de cette étude sont principalement destinés aux réseaux WSAN, puisqu'elle correspond à la forme automatisée ou semi-automatisée des applications proposées dans la littérature. L'apport théorique de l'étude des longueurs de chemins est complété par une expérience réelle sur réseau de capteurs, dont les résultats confirmeront, outre la validité de l'approche chorégraphiée dans l'inclusion des WSAN dans l'Internet des objets, la conformité des mesures par rapport aux bornes données par la théorie.

Le 4e chapitre s'applique à résoudre la problématique de la diversité des outils à l'œuvre dans l'Internet des objets. Multiples matériels, besoins de reprogrammation rapide, accès distants, ces différents items sont traités par l'introduction d'une machine virtuelle, D-LITe, pilotable à distance grâce à une approche REST, et offrant un accès abstrait aux capacités de la machine. Cet ensemble représente la deuxième contribution de cette thèse. Les conditions de l'implémentation de D-LITe, le langage de programmation associé SALT, ses différents formats et ses fonctionnalités étendues y sont traités. Les applications chorégraphiées étant sujettes aux erreurs, ce chapitre se clôt sur une description de leurs effets possibles sur notre solution, et propose une construction d'arbres de vérifications en cascades dont l'objectif est d'en contenir les conséquences. Un article de synthèse décrivant le mécanisme de cette extension de D-LITe est en cours de soumission.

La simplification de la création d'une application IoT constitue la 3e contribution de cette thèse et fait l'objet de notre 5e chapitre. La composition graphique d'une chorégraphie grâce à un *mashup* est soumise à un ensemble de règles et de contraintes pour en vérifier la faisabilité. Nous nous intéressons ici à définir ces contraintes, à les modéliser, puis à proposer des processus de validation de l'ensemble. Ainsi, il devient possible de proposer une solution participative afin que divers intervenants réalisent, normalisent, puis partagent des morceaux de code que des utilisateurs pourront combiner à volonté. L'accès au processus de création d'une application est simplifié, rapide, et son évolution facilitée par l'approche très intuitive du mashup. L'ensemble permet pourtant une construction robuste grâce au modèle décrivant les relations entre éléments et les procédures de vérification que le système est capable d'effectuer.

Le dernier chapitre conclut ce travail et introduit des idées et perspectives nouvelles qui pourront succéder à cette thèse.

Chapitre 2

État de l'art

Sommaire

"Je vous poursuivrai en justice", dit la porte.

Ubik
PHILIP K. DICK

Les évolutions majeures ont parfois des visages anodins. Dans son article visionnaire et fondateur "*The Computer for the 21st Century*"[133], Mark Weiser dessine en 1991 les contours de l'Internet des objets (IoT) en ces termes : "*Les technologies les plus importantes sont celles qui disparaissent. Elles tissent elles-mêmes le tissu de notre vie quotidienne au point de s'y confondre*".

Parce qu'elle s'immisce dans les objets du quotidien, l'informatique est en train de transformer notre rapport à l'environnement. S'émancipant de l'appareil qui l'hébergeait et lui était dédié (l'ordinateur), le *traitement automatique de l'information*[1] se propage dans une large gamme d'appareils, se dissémine toujours plus, dotant de prémisses de raisonnement ce que l'on finit par appeler des "*objets*", terme très générique, à l'image de cet assemblage d'éléments hétéroclites qui nous entourent, si divers, nombreux, multiformes et polyvalents. Cette extension des usages et de la présence d'une "*intelligence*" automatique dans les éléments et les endroits les plus variés, incarne la vision de Mark Weiser [133], qu'il avait lui même nommée "*ubiquitaire*", dans laquelle l'informatique devient partie intégrante bien qu'invisible de la vie des utilisateurs.

A l'image du smartphone, et sans que cela se limite aux outils principalement électroniques, les éléments de notre environnement (certains auteurs le regroupent sous le terme *Choses* afin d'en montrer la grande diversité, insistant sur cette indéfinissable classification) deviennent "*smart*", capables de calculs, de l'exécution d'applications, de stockage de données et d'accès au réseau sans fil. Alors s'ouvre la possibilité d'utiliser leurs capacités pour inventer des usages innovants, d'étendre notre contrôle par de nouvelles fonctionnalités, et même imaginer des interactions inédites entre objets. Des services d'"*environnement augmenté*" deviendront disponibles, qu'il s'agisse du véhicule qui trouve automatiquement une place de parking libre, de la gestion complète de l'habitat, de services d'urgence personnalisés et contextualisés en cas de catastrophe, d'assistance aux personnes âgées, ou du déclenchement de fonctionnalités offertes par un appareil public et à proximité, en rapport à nos besoins immédiats. Omniprésente, interagissante avec l'utilisateur, ou avec ses objets, l'approche "*pervasive*" met l'informatique à la périphérie de la conscience de l'utilisateur, son attention étant de moins en moins mobilisée grâce à un intégration sans effort, transparente, de la dimension informatique de son milieu, ce que M. Weiser définit comme étant le "*Calm Computing*" [134].

Dans cette partie, nous nous attacherons à décrire les fondements sur lesquels est bâti l'Internet de objets. La réalisation d'un système fédérant différents composants demande une analyse des différents enjeux, freins et atouts de chacun des éléments qui le constituent. Cet état de l'art porte à la fois sur les matériels, et notamment les réseaux de capteurs qui représentent une part importante (bien que non-exclusive)

1. Définition usuelle de l'informatique.

de l'Internet des objets (puisque par définition, la base même de ce domaine réside dans l'interpénétration des différentes technologies de l'informatique), mais aussi des services de l'Internet qui constituent la partie la plus visible et structurante de l'ensemble, et des différentes solutions qui sont proposées à ce point de jonction qu'érige l'IoT.

2.1 Les capteurs, composants majeurs de l'Internet des objets

L'essor des capteurs suscite un grand intérêt aussi bien dans la communauté de la recherche que dans le monde industriel [45] [106]. Les apports attendus de cette branche active, à la croisée de l'électronique, du réseau et de l'informatique, permettent d'envisager une perception améliorée d'un milieu. Cette technologie permet une évaluation chiffrée de l'environnement direct, des informations localisées et des traitements informatiques. L'ensemble des mesures doit être mis à disposition de l'utilisateur, et lui être transmis, ou du moins être accessible à distance, offrant alors une véritable valeur ajoutée. Pour y parvenir, ces objets de petite taille, à coût très réduit, pourront s'organiser entre eux et, pour peu qu'ils s'astreignent au respect de standards clairs et robustes, fournir l'accès le moins contraignant possible à toute personne/service désireux de l'utiliser.

2.1.1 Les capteurs

Les capteurs sont de petits appareils disposant de capacités de mesures, voire d'actions, sur leur environnement. La température, le taux d'humidité, la luminosité ambiante, la détection de présences ou de mouvements via un accéléromètre, présence de gaz, de polluants ou encore la géolocalisation font partie des informations les plus couramment collectées sur ce type de matériel (figure 2.1). Selon les capteurs, ou grâce à l'ajout de cartes additionnelles, la pression atmosphérique, le niveau de radiation ou la pression acoustique (événements sonores) peuvent aussi être quantifiés. Les spécificités innovantes de ces capteurs résident dans leur taille et leur coût réduit, tout en étant dotés des capacités de traitements de l'information, et des possibilités de transmission sans fil [20] [49] [66] [106] [136].

La nature de l'objet "*intelligent*" a ceci de particulier qu'il dispose de possibilités de calcul et de transmission des données. L'échange d'informations, voire l'interaction entre différents objets est envisageable, et ce même sans intervention des utilisateurs. L'objet va "*capter*", mesurer une caractéristique physique de son environnement, éventuellement lui appliquer un traitement informatisé, et fournir le résultat à d'autres (utilisateur, ordinateur, etc.).

Si les capteurs disposent d'éléments pour évaluer une caractéristique physique de leur environnement, certains d'entre eux peuvent également agir sur cet environnement : on parle alors d'*effecteurs* [19] (I. Akyikdiz les appelle ici "*actors*"). Ces *effecteurs* sont souvent plus puissants en terme de capacités mémoire, de traitement,

Figure 2.1 Panorama des capteurs utilisés dans le laboratoire LIGM

voire en réserve d'énergie. Ils sont aussi moins nombreux que les capteurs, car pour des raisons pratiques, il est cohérent de disposer d'un grand nombre de points de mesures tandis qu'il vaut mieux restreindre le nombre de points d'actions, et éviter ordres et contre-ordres incohérents. Toutefois, cette approche tend à être modifiée par l'évolution des WSANs (*Wireless Sensors and Actuators Networks*) lorsque ceux-ci s'intègrent dans l'Internet des objets [90]. Dans ce cadre, leur usage diffère [39] pour s'orienter vers le service, et la multiplicité des capteurs s'amenuise (dans une vision domotique, on n'utilisera qu'un nombre réduit de capteurs de température, un par pièce par exemple).

Certains des capteurs dont nous disposons pour nos expérimentations symbolisent des effecteurs, puisqu'ils permettent au choix l'émission de sons via leur haut parleur, ou l'allumage des diodes de différentes couleurs dont ils équipés[2]. Sur la figure 2.1, les caractéristiques des matériels présentés donnent un panorama des actions et mesures assez diversifiées que ces outils réalisent. Le TelosB qui y figure nous a permis de développer des applications IoT grâce à ses 48 Ko de mémoire morte, ses 10 Ko de mémoire vive (hébergeant notre code et l'ensemble du système d'exploitation) et son processeur 16 bits TI MSP430 cadencé à 8 Mhz, chiffres qui donnent une idée des limites du matériel. Les différents capteurs et effecteurs du TelosB procurent les bases suffisantes pour valider des expérimentations. De son coté, l'Arduino se révèle être un autre exemple d'appareil très évolutif, grâce l'offre très fournie de son système d'extensions (*shield*). La possibilité d'agir sur un relais qui piloterait toutes sortes d'appareils est tout à fait envisageable, offrant alors aux applications une main-mise plus importante sur leur environnement. L'effecteur contrôlerait ainsi la mise en route, le pilotage ou l'arrêt de n'importe quel équipement électrique, autorisant son contrôle et, éventuellement, sa configuration à distance.

2. Nous sommes aussi capable d'émettre un SMS avec un smartphone, de le faire vibrer, d'activer son appareil photo.

Les outils présentés figure 2.1 sont principalement à destination de la recherche, mais leurs fonctionnalités opérationnelles (bouton, led, capteurs de température et lumière, buzzer...) suffisent pour tester des architectures réalistes.

2.1.2 Les réseaux de capteurs

Afin de satisfaire les besoins de communication entre eux, les capteurs sont équipés de dispositifs sans fil pour l'émission et la réception de données. Cela ne suffit cependant pas à rendre un ensemble de capteurs accessibles, ou du moins de manière inter-opérable, transparente et simplifiée. Pour cela, les capteurs doivent aussi s'organiser. Ce qui caractérise un réseau de capteurs, c'est que ses éléments sont de très petits appareils, dotés de capacités de transmission sans fil, autonomes en énergie et dont le positionnement est, le plus souvent, libre (dans le sens où il n'est pas toujours contrôlé, anticipé, volontaire, parfois confié au hasard, par dispersion, par exemple) [136]. Un réseau de capteurs peut être plus ou moins étendu, la concentration d'éléments et leur type très variables, et les nœuds habituellement figés à leur position d'origine. Toutefois, M. Younis [137] présente un possible re-positionnement des nœuds, soit pour améliorer l'efficience du réseau, soit pour gérer a posteriori des contraintes apparues (c'est-à-dire voulues, édictées par l'utilisateur par exemple) ou détectées à l'usage (c'est-à-dire indépendantes de sa volonté, pannes, extinction ou brouillage par exemple). S. Hashish et al. décrivent les économies d'énergie apportées par des repositionnements pertinents et répétés du seul puits [74]. Les capteurs, en économisant leurs batteries, doivent être capables de fonctionner plusieurs années [106]. Cette exigence fait partie de leur cahier des charges, et le réseau auquel ils participent doit de toute évidence prendre en compte cette contrainte [20]. Le réseau de capteurs devra souvent s'organiser de façon autonome, car là encore, la nature même des nœuds participants, l'objectif de leur conception, leur déploiement sans contrainte préalable ni anticipation de leur future utilisation, induisent un comportement dynamique et souple de la structure du réseau [17] [54].

Les informations collectées dans un réseau de capteurs convergent vers un puits de données (le *sink*) chargé de la collecte des mesures. Les capteurs doivent donc utiliser la structure de leur réseau sans fil afin d'atteindre ce puits, qui n'est peut être pas, pour une majorité d'entre eux, directement accessible en un saut [45]. Le rôle du puits est la récupération de l'ensemble des valeurs et leur communication vers l'extérieur. Le puits joue à la fois le rôle de passerelle, mais peut aussi avoir, dans certains cas, une action non négligeable dans l'organisation de la structure du réseau. Dans les constructions à base d'arbre par exemple, il en est souvent la racine [12] [127]. De par son rôle spécifique, le puits sera souvent sur-dimensionné par rapport aux autres nœuds du réseau (énergie illimitée, puissance de traitement supérieure, etc.) [140] [136].

2.1.3 Les contraintes spécifiques aux capteurs et réseaux de capteurs

Pour des raisons de coûts et de taille, les capteurs sont des équipements électroniques soumis à de considérables contraintes [66]. Le domaine d'application de ce type de produits impose un très faible coût de fabrication tout en exigeant une durée de vie importante et l'impact le plus ténu possible sur l'environnement, aussi bien en terme de taille que de pollution. Mais ils doivent aussi proposer des capacités de traitement, de mesures du milieu ambiant, de transmission et d'organisation du réseau [106]. L'empreinte du capteur sur le milieu doit rester la plus faible possible, et donc n'imposer aucun câblage, ni apport extérieur en énergie, etc [136]. L'objectif des concepteurs est d'offrir la mise à disposition d'une multitude d'éléments redondants (du moins dans l'approche traditionnelle des réseaux de capteurs), aussi bien en termes de mesures que de relais de messages et de traitement de l'information.

2.1.4 L'énergie

La plus importante contrainte à laquelle sont soumis les réseaux de capteurs concerne l'énergie. L'autonomie temporelle des nœuds s'évalue en termes d'années (de 5 [106] à 10 ans ou jusqu'à 3 ans selon les cas [136]), tout en étant un objet actif, disponible, communiquant, traitant des données, et partie prenante dans l'organisation du réseau. Pourtant, l'accès physique à l'objet et sa maintenance sont souvent inenvisageables (soit pour des raisons d'inaccessibilité, si ceux-ci sont par exemple chargés de surveiller les incendies dans la forêt, et donc disséminés par la voie des airs au dessus de la zone à surveiller, ou coulés dans du béton dans le but de surveiller l'ossature d'un bâtiment, ou tout simplement parce qu'il semble peu cohérent d'avoir à intervenir sur un grand nombre d'appareils prétendument autonomes) [49]. En tout état de cause, ces impératifs impliquent une grande vigilance dans la conception des outils qui exploitent les données mesurées ou accèdent aux nœuds, aussi bien dans la brièveté et la concision des échanges que dans leur fréquence, les méthodes d'interrogation utilisées, l'emploi du réseau ou le mode de diffusion des résultats obtenus [30] [114].

Pour parvenir à une gestion économe de l'énergie, les chercheurs sont intervenus à tous les niveaux : que ce soit dans les choix qui ont gouverné la création de protocoles réseaux spécifiques (en restreignant de manière drastique le débit au profit d'une consommation moindre), ou en terme d'organisation, d'adressage, de routage, de mode d'accès au médium, c'est tout d'abord le coût énergétique qui est évalué. Endormissement du nœud, préférence pour des échanges nécessitant le minimum de questions/réponses, contenus condensés, voire se condensant entre eux, à la volée, compression de la mise en forme, format d'adressage court, tout est repensé avec, en ligne de mire, une consommation minimum d'énergie. Un panorama des contraintes et de leurs solutions recensées est présentée dans la taxinomie proposée par G. Anastasie et al. [23] [30].

L'estimation de la consommation d'énergie et la recherche d'approches appropriées pour optimiser la durée de vie des WSAN est au cœur de nombreuses analyses et propositions de solutions [139] [47] [89] [66]. Au niveau applicatif, dans lequel cette thèse s'inscrit, nous retiendrons principalement que ce sont généralement les fonctionnalités de transmission et réception sans fil qui sont les plus consommatrices. Comme le préconisent L. Mottola et G.P. Picco [96] ainsi que G. Meyer et. al. [93], nous nous orienterons vers une solution dans laquelle le contrôle de la logique de l'application est concentrée à l'intérieur du réseau, la décision faite au plus proche des événements qui en sont la source, afin de limiter les échanges, réduire le coût énergétique et la latence.

2.1.5 La puissance de traitement

Dans les réseaux de capteurs, et plus encore dans l'Internet des objets, chaque nœud doit être un minimum "*intelligent*", c'est-à-dire capable de traiter, de modifier, de combiner des informations, tout en offrant des accès à ces informations.

Puisque le coût [3] est l'un des principaux facteurs qui président à la fabrication de ces objets [19], les concepteurs limitent les capacités de traitement de l'information. La fréquence des processeurs utilisés est assez faible, parfois modifiable dans le temps (pour diminuer la consommation énergétique), et la taille de l'espace mémoire très réduite. A titre d'exemple, le TelosB de la figure 2.1 dispose d'un processeur 16 bits cadencé à 8 Mhz, et d'un espace mémoire total de 58 Ko. L'Imote2 contient un processeur 32 bits qui, en mode économie, ramène sa fréquence à 13 Mhz, et offre un espace mémoire de 256 Ko, l'Arduino quant à lui utilise, selon les versions, un processeur 8 bits à 20 Mhz et 32 Ko de ROM, ou un processeur 32 bits à 84 Mhz et 512 Ko de ROM. Malgré la disparité de leurs puissances, ces objets doivent collaborer et, impliqués dans une même application, offrir une large gamme de fonctionnalités compatibles entre elles.

Cette approche est d'autant plus prégnante, à notre sens, qu'elle découle de l'inclusion des WSAN dans le domaine plus large de l'Internet des objets, domaine qui demande une implication supérieure de chacun des participants. Il faudra donc chercher à maximiser l'utilisation des ressources que les facultés restreintes des objets offrent, afin de présenter des services aux multiples usages possibles. Ces caractéristiques auront des implications importantes dans les solutions que nous proposerons à plus haut niveau dans la conception du système.

2.1.6 Les échanges sans fil

Les technologies sans fil sont maintenant bien connues et répandues dans l'informatique et la communication. Les réseaux de capteurs bénéficient de protocoles qui leur sont dédiés. Nous nous sommes limités à l'utilisation de 802.15.4 [14], particuliè-

3. À titre d'information, le prix de ces objets se situe dans un intervalle de 20 à 80 euros maximum.

rement adapté aux contraintes énergétiques, mais qui offre un débit faible (250 kb/s maximum) et une taille de trame réduite [138].

Les réseaux sans fils fournissent un usage transparent pour l'environnement matériel, et c'est leur principal atout : pas de câblage, pas (ou peu) d'infrastructure nécessaire, des possibilités de nœuds mobiles, la liberté d'accès et des évolutions (en termes de nombres de nœuds, d'espace ou d'étendue du réseau) peu contraignantes en termes d'impact physique sur le milieu. Cependant, dans ce type de réseaux sans infrastructure, l'absence d'un cadre organisé oblige à concevoir des systèmes s'auto-organisant [54], afin de pallier l'absence d'intervention humaine, raisonnée, planificatrice. Il s'agit donc d'imaginer des mécanismes automatiques permettant aux nœuds de découvrir leurs voisins, ou la structure existante, de se configurer selon celle-ci, et d'élaborer une construction autonome capable d'assurer la collaboration entre tous [17][45] [136] [99] [115] [19]. Ces assemblages peuvent par exemple être conçus sous la forme d'une hiérarchie construite à la suite à d'échanges d'informations, ou suivant des rôles prédéfinis, ou encore grâce à une association de nœuds se considérant d'égal à égal.

2.2 Les services de l'Internet et l'IoT

Dès lors que les différents nœuds communiquent, les informations dont ils ont la charge, ou les actions qu'ils peuvent accomplir deviennent accessibles. Cependant, l'accès à celles-ci n'est pas forcement pratique. Dans quelle mesure son usage est-il connu, décrit, et selon quel mode opératoire ? Qui peut y accéder, et comment ? Dans un objectif d'utilisation simplifiée, voire automatique, ou même pervasive, quelles méthodes sont envisageables pour franchir le pas entre la possibilité d'un accès à une ressource offerte par le système, et son utilisation tangible et pragmatique ?

Nous proposons tout d'abord un panorama des systèmes qui permettent de décrire les services disponibles dans un environnement ouvert et diversifié tel que celui que peut rencontrer un utilisateur, puis nous nous intéresserons aux mécanismes proposant de faire interagir ces services entre eux.

2.2.1 L'approche services SOA

SOA (*Services Oriented Architecture*) est une approche de l'Internet des PC dans laquelle l'ensemble des informations, des traitements, et des ressources du système d'information sont présentées sous la forme de services [59]. Il ne s'agit pas à proprement parler d'une norme, mais plutôt d'une architecture, d'une construction assemblant les technologies existantes. La finalité de cette démarche de conception est de parvenir à l'interopérabilité entre les outils et solutions employées [39], qui doivent être normalisés et les plus répandus possible (dans le cas contraire, cette inter-opérabilité serait plus difficile a assurer).

Plusieurs organismes, comme l'OASIS [8], se consacrent à la production et description de normes garantissant une standardisation des constituants SOA. L'objectif de cette démarche est de rendre possible une composition, un assemblage

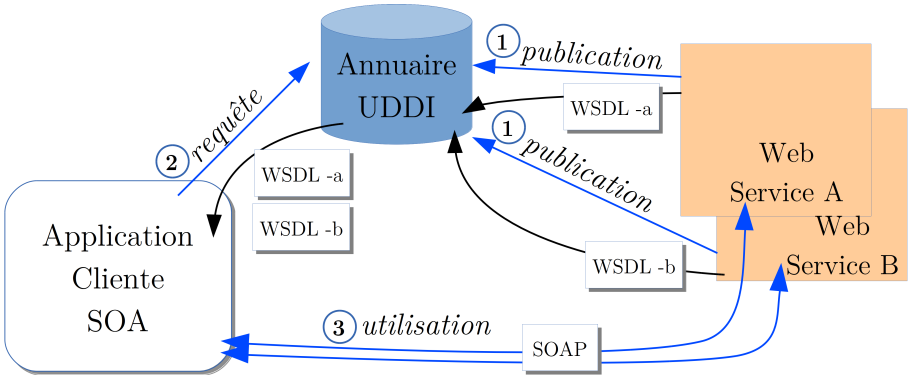

Figure 2.2 L'Architecture Orientée Service (SOA). Les services web sont recensés dans un annuaire UDDI qui stocke la description WSDL du service offert. Les applications clientes, après avoir identifié le service recherché, peuvent l'invoquer grâce au format SOAP.

de ces services afin de construire de nouvelles applications. Plutôt que d'avoir des applications monolithiques, et d'ensuite concevoir des passerelles entre ces différents traitements (sous forme d'export, par exemple, ou tout autre mode de conversion), la démarche SOA opte pour l'utilisation de briques logicielles, accessibles, ouvertes, décrites, offrant des services atomiques [101]. L'architecture ainsi réalisée est alors *découplée*, c'est-à-dire que les différents éléments échangent de l'information tout en étant très peu dépendants les uns des autres, uniquement liés par le contrat stipulant la façon de communiquer [129]. On obtient alors une solution faiblement couplée, chaque élément pouvant évoluer sans remettre l'édifice en cause, pour peu qu'il continue à respecter la même interface visible.

L'architecte chargé de la construction d'une application va composer avec ces services (figure 2.2). Pour parvenir à ses fins, cet architecte demande à ce que les services soient décrits de manière standard dans un annuaire (figure 2.2), et proposent des formats d'échanges (ou d'encapsulation de l'échange) suffisamment riches, précis, soumis à des règles à la fois strictes mais facilement évolutives, pour permettre l'exploitation des ressources offertes. Ce que chaque service fournit sera documenté (souvent sous la forme XML), ce qu'il attend, en termes de paramètres, décrit de la même façon en WSDL (*Web Services Description Language*, un langage descriptif au format XML) [7]. L'échange lui-même est aussi présenté en XML (par exemple via le protocole SOAP [9] pour l'appel au service et pour le contenu échangé).

Autant cette démarche a du sens dans la construction d'un service qui sera offert à de très nombreux utilisateurs, autant dans le cadre de l'Internet des objets les rapports évoluent, et l'organisation est amenée à être repensée. Les applications Internet des objets mettent en jeu des éléments personnels, doivent répondre à des problèmatiques individuelles (*home automation*, ou *context-aware* dans les villes intelligentes). Alors que l'intervention d'un architecte est requise pour la construction d'un agrégat solide de services à destination d'une population d'utilisateurs la plus large possible, chaque application IoT ne sera peut être utilisée que par son concepteur et ses proches, car adaptée plus spécifiquement à son usage, et/ou à certains

de ses objets. Dans l'Internet des objets, le circuit concepteur/consommateur est réduit, et parfois ces rôles sont tenus par une même et unique personne.

2.2.2 Les services web

Les services web (*Web-Service*) [11] sont la déclinaison principale de l'architecture SOA utilisant le protocole HTTP. Si un nœud doit offrir un tel service, pourquoi ne pas utiliser un protocole répandu, bien connu et robuste ? Habituellement, le *World Wild Web*, tel qu'il a été conçu par Tim Berners-Lee, permet l'accès d'un utilisateur humain au contenu d'une ressource stockée sur une machine distante. HTTP [32] décrit l'établissement de la connexion TCP vers cette ressource, décline un ensemble de paramètres pour spécifier la ressource voulue (l'URI), et détermine la façon dont le serveur va répondre (entête HTTP) avant que celui-ci ne ferme la connexion (sauf dans le cas du keep-alive de la version 1.1 du protocole [61]). Le contenu de l'échange, sous forme de texte, est souvent encapsulé dans un format *html* afin d'agrémenter sa présentation pour le navigateur de l'internaute.

Les services web sont aux applications clientes ce que le Web est aux utilisateurs [34]. Ici, on utilise ce même protocole applicatif, mais dans l'idée d'offrir des informations adaptées à leur utilisation par d'autres programmes, et non plus par un être humain. Les besoins en termes de présentation de l'information et de sa granularité sont spécifiques. Souvent, une application a besoin d'une simple donnée, afin de la traiter (un horaire, un prix, un nom, une url...). La présentation qui lui importe concerne plutôt le typage de la donnée et son encodage, et non un format de mise en page.

C'est la raison pour laquelle les échanges réalisés entre *Web-Services* utilisent le format SOAP (initialement l'acronyme de *Simple Object Access Protocol*). Ce protocole, décrivant comment invoquer l'opération sur le serveur distant, opte pour le format XML qui type et structure les contenus. Souvent l'échange ne concerne qu'un petit nombre de données à chaque salve (quitte à multiplier les services offerts et/ou les accès à ces services). Parmi les services web publics, nous pouvons citer ceux de Google qui permettent à une application d'interroger le service du moteur de recherche et de récupérer ses réponses, réponses qui seront exploitées directement dans l'algorithme... Ou encore les services offerts par les entreprises de transport de passagers (par rail, par air...) pour l'interrogation, voire la réservation de places, le tout sous une forme adaptée à son traitement par une application informatique (sous forme d'arbre XML). La conception d'une nouvelle application de type web (ici à destination des êtres humains) combinant ces services, et ceux d'autres serveurs, est alors facilitée et plus souple. Elle pourrait par exemple agréger des données concernant des chambres d'hôtels, et l'application composerait, organiserait l'ensemble de ces informations pour proposer des voyages clef en main.

Les apports, en termes de versatilité de la structure du contenu de ces échanges, introduisent une grande flexibilité de composition des différents services [129]. Cependant, leurs nombreux messages de taille importante impliquent des débits réseaux, des capacités de traitement et de stockage mémoire et une énergie consé-

quents chez l'ensemble des participants. L'application à l'identique de ces solutions à l'Internet des objets ne peut se faire sans adaptation aux contraintes de certains de ses constituants les moins puissants.

2.2.3 L'approche REST

Representational State Transfert (REST) a fait l'objet de la thèse de R. Fielding [62], auteur qui a aussi participé à la définition du protocole HTTP. Selon lui, le concept de SOA propose une idée intéressante pour une approche orientée services du Web. Cependant, le protocole principalement utilisé dans ces architectures (SOAP) pose problème. En effet, les messages SOAP sont des flux XML de taille non négligeable, qui sont transmis entre le client et le serveur par le protocole HTTP, protocole applicatif (donc normalement à usage "final").

Or, ici, HTTP n'est utilisé que dans le but de transporter le message SOAP [142]. En fait, il joue le rôle de la couche de transport. HTTP n'a pas vocation à n'être qu'un vecteur d'échange, un simple porteur de message pour un protocole qui lui serait supérieur (même si le fait qu'il permette l'accès à des contenus html semble le limiter à un simple rôle de transporteur). SOAP n'utilise HTTP que pour échanger de l'information, tandis que toute la dynamique de l'échange va être décrite par SOAP. SOAP a le gros défaut d'être particulièrement verbeux, ce qui est rédhibitoire dans les réseaux de capteurs. De plus, HTTP offre déjà tout un ensemble de mécanismes mis en œuvre dans SOAP. HTTP peut implémenter CRUD (*Create, Retrieve, Update, Delete*) [97], les opérations atomiques de base pour la gestion des données. HTTP sait typer les données (*Content-Type*), et les entêtes (*headers*) dont il dispose fournissent tout le contexte requis autour des données transmises. HTTP permet d'accéder à des ressources spécifiées dans l'URL. Tout cela revient à dire que SOAP, ou une grande partie de ce qu'il apporte, peut être réalisé grâce aux mécanismes proposés par HTTP [62].

REST n'est pas un protocole, mais plutôt une approche architecturale, dont l'objectif est de faciliter la programmation d'applications orientées services en utilisant HTTP. REST met l'accent sur les rôles de client et de serveur, sur l'identification des ressources accessibles grâce à un adressage unique, et sur des échanges ne nécessitant pas d'états, donc de suivi de l'ensemble des actions et leur contextualisation. Les échanges doivent être auto-suffisants et ne pas dépendre d'actions préalables ni peser sur les réactions futures. Aussi, la faculté d'HTTP de transporter des informations, d'en proposer le typage et la description, et de disposer des verbes nécessaires à CRUD autorise la gestion distante d'éléments et l'invocation de commandes de manipulation de ces éléments.

Dans le cadre des capteurs et des environnements très contraints en général, il semble pertinent de disqualifier SOAP, notamment par sa lourdeur, au regard du service qu'il est censé rendre. Cette critique qui a été conçue pour les environnements réseaux plus puissants de l'Internet s'impose d'autant plus pour les réseaux de capteurs [105]. La plupart des opérations courantes peuvent être réalisées grâce à HTTP, plus simplement, de façon plus légère et concise [68]. La concision et la

légèreté des échanges sont précisément ce que les capteurs requièrent, au regard de leurs contraintes énergétiques.

2.2.4 La chorégraphie et l'orchestration de services

La disponibilité de services web chargés de la réalisation de tâches atomiques, disséminés sur le réseau, flexibles et autonomes permet d'en imaginer des combinaisons, dans le but de fabriquer un processus complet. La construction d'une telle combinaison s'appréhende de deux façons différentes, décrites dans l'article de C. Peltz [102]. D'un coté la conception d'un algorithme sous le contrôle d'un point central, chargé de l'enchaînement des appels de services et de la gestion des résultats retournés, ou *orchestration* de services. A contrario, une approche dans laquelle chaque service interagit avec les autres sans être placé sous la responsabilité d'un nœud principal est dénommée une *chorégraphie* de services.

La vue orchestrée de la conception d'une application reste assez conforme aux démarches des programmeurs, dans laquelle un programme principal est chargé du "*flux de contrôle*", et multiplie les appels à des ressources (qui sont ici les services web). BPEL (norme IBM décrite dans [51]) est le langage le plus couramment utilisé pour modéliser des orchestrations.

La chorégraphie quant à elle met plus l'accent sur les relations entre les différents services, c'est-à-dire les causes, les dépendances et les exclusions. La chorégraphie ne décrit pas d'effets internes (alors que l'orchestration au contraire se concentre sur le raisonnement interne du nœud principal) mais plutôt les réactions de chacun des services soumis à des stimuli échangés [27]. Dans cet article, A. Barros et al. décrivent le déroulement de différents services en parallèle, chacun recevant des messages d'un autre, réagissant en progressant dans son propre flux de contrôle, puis déclenchant une action à son tour. Les auteurs présentent le langage WS-CDL (*Web-Service Choreography Description Language*), qui est décrit plus en détail par la norme W3C [81].

Ces deux approches sont suffisamment différentes pour qu'on s'interroge sur leurs recouvrement : est-il possible d'exprimer les mêmes règles et de satisfaire aux mêmes besoins ? L'étude de la "*conformité*" d'une même application réalisée selon les deux approches a été étudiée par N. Busi et al. dans [41], qui proposent une formalisation du modèle d'expressivité de la chorégraphie et de l'orchestration. Leur conclusion est que le recouvrement n'est pas toujours obtenu, mais qu'il est tout à fait possible de résoudre les mêmes problématiques avec ces deux modes d'organisation. Q. Zongyan et al. [108] ont aussi exploré les caractéristiques de WS-CDL au travers d'une version modélisée qu'ils ont appelée "*Chor*". Ils font apparaître un besoin de "*rôle dominant*" que doit pouvoir endosser un service afin d'améliorer l'expressivité de la chorégraphie et couvrir certains besoins de synchronisation et de boucles. Grâce à cette notion, la structure de contrôle alternative (*switch*), ou les boucles, peuvent être exprimées dans une chorégraphie.

Enfin, une étude de M. zur Muelhlen et al. [142] s'intéresse à l'utilisation de la chorégraphie dans le cadre de SOAP et de REST, et compare leurs points forts

Figure *2.3* Quelques objets intelligents mis en œuvre dans l'Internet des objets. Des capteurs et effecteurs, en passant par un smartphone, un réfrigérateur connecté, et le Nest (objet emblématique de l'IoT), thermostat communicant.

et faibles face aux fonctionnement de la chorégraphie. REST fournit un couplage faible, et la clarté de la définition des opérations voulues qui apparaissent dans le message, tandis que SOAP permet de décrire la complexité de certaines opérations et propose un suivi des échanges, intéressant notamment pour déboguer la cinématique du dialogue.

Nous pensons que l'approche chorégraphiée est adaptée à l'Internet des objets [93]. Elle permet à la fois d'alléger le recours aux ressources des réseaux WSAN [96] qui participent à l'IoT. La chorégraphie respecte les contraintes énergétiques, car les interactions directes entre nœuds sollicitent moins le réseau qu'une remontée des données à un point central. De plus, les capacités de traitement ajoutées aux objets (certes limitées, mais bien présentes) permettent l'implémentation des algorithmes décrivant la tâche à effectuer sur l'objet lui-même [71].

2.3 Les protocoles à l'œuvre dans l'Internet des objets

L'Internet des objets est à la croisée d'un ensemble de technologies [25]. C'est la démocratisation et l'accès à de multiples sources de données et traitements par des moyens de plus en plus transparents (sans fil, automatisé, auto-négocié) qui permettent d'en imaginer la forme. Les capteurs ont un rôle non négligeable à jouer dans ce domaine, car ils sont les objets qui vont se relier, grâce à l'Internet, à l'ensemble des outils dont nous sommes familiers [31]. Cependant, l'Internet des objets ne se limite pas à ceux-ci, et intègre une plus vaste gamme d'éléments (figure 2.3), aussi bien provenant de la domotique, que de l'Internet de PC traditionnel. Pour réaliser des applications IoT, la connectivité offerte par ces nœuds doit être exploitable à un plus haut niveau. Il est donc nécessaire d'à la fois organiser le réseau et la connexion à ce réseau, mais aussi proposer un accès efficace et exhaustif aux services mis à disposition.

2.3.1 DPWS

Historiquement, c'est UPnP (*Universal Plug'n Play*) qui a permis l'utilisation pragmatique et simplifiée des différentes ressources, périphériques, dont un utilisateur dispose, notamment dans le cadre familier de son domicile. UPnP est un ensemble de protocoles et une architecture ouverte gérée par le Forum UPnP [5]. Avec lui, les différents appareils présents se découvrent entre eux, présentent leurs services, et deviennent utilisables sans que cela n'exige de l'utilisateur une expertise technique poussée, ni d'intervention de configuration, de recherche, ou d'interconnexion du matériel longue et/ou complexe. Une version de ce protocole, DLNA [4], est spécialisée dans l'interconnexion des différents appareils multimédias. DLNA permet à des équipements de se découvrir et d'exploiter leurs services aisément, même si leurs fabricants sont différents (c'est tout l'intérêt du protocole pour l'utilisateur).

Ce protocole dont la raison d'être semble assez ambitieuse (faire tout fonctionner facilement est un objectif souvent annoncé, rarement atteint) obtient d'assez bons résultats et parvient à tenir ses promesses. Cependant, cette réussite masque de graves lacunes en terme de sécurité des accès, car l'auto-configuration et l'utilisation quasi-automatique autorisent des usages frauduleux et non désirés des ressources gérées, échappant au contrôle de leur propriétaire [116], compromettant alors l'extension du protocole vers d'autres domaines.

DPWS (*Device Profiles for Web Services*) [15] est présenté comme la version 2.0 d'UPnP, sous le contrôle de l'organisme de certification OASIS [8]. L'objectif est le même : offrir un accès facile et intuitif aux ressources. DPWS offre une palette d'outils supplémentaires, embarque un système de sécurité, un bus logiciel, des possibilités de publication/souscription. Comme pour UPnP, les différentes ressources s'auto-configurent, se découvrent, présentent leurs services, décrivent les conditions d'accès et le format des valeurs renvoyées. Les échanges utilisent le format SOAP [9], le contenu des messages est encapsulé dans un format XML très descriptif, bien-formé et valide (c'est-à-dire se référant précisément à un format particulier et rigoureusement défini, dont l'accès sera obligatoirement donné dans l'échange, sous forme de lien).

2.3.2 Problèmatique spécifique aux réseaux de capteurs

L'approche services est de plus en plus acceptée comme solution efficace pour une utilisation plus "applicative" des réseaux de capteurs [124]. Le portage de DPWS sur les réseaux de capteurs a fait l'objet du projet européen SODA [10]. Comme dans sa version générique, les capteurs se découvrent, présentent leurs services, s'enregistrent éventuellement les uns auprès des autres, puis s'échangent de l'information. Concevoir des applications mettant en œuvre ces capteurs devient alors possible. Nous avons déjà présenté plus haut les contraintes des capteurs. La plus importante concerne l'énergie, et toutes les autres en découlent. Les choix concernent principalement une limitation de la capacité de traitement, un espace mémoire très limité, une couche réseau à débit très faible et une taille de charge utile limitée. Ces carac-

téristiques vont peser de façon non négligeable sur le portage d'une architecture à base de services sur ces équipements. En effet, les architectures SOA ont été conçues pour des réseaux dans lesquels les débits sont importants, et qui utilisent des protocoles réseau dont les trames supportent des charges utiles conséquentes, et dont les équipements disposent de confortables ressources en mémoire, en capacité de traitement, et d'une énergie illimitée.

Or la simple implémentation de TCP sur un capteur pose des problèmes [52], car toute connexion initiée doit être suivie, et donc disposer d'une empreinte en mémoire centrale, mémoire qui fait défaut au capteur. Imaginer pouvoir installer un serveur HTTP, basé sur TCP, échangeant des messages SOAP dont la taille "à vide" est telle qu'ils consomment plusieurs trames 802.15.4 [4] à eux seuls, tout cela plaide pour une importante ré-évaluation des protocoles applicatifs utilisés dans une approche services pour les réseaux de capteurs [44]. La simple implémentation d'un serveur HTTP pose problème, car l'utilisation de TCP sur des architectures si limitées en mémoire est quasi-proscrite. La mise à disposition d'un service web à contenu dynamique transporté dans des messages XML de type SOAP telle qu'elle est organisée dans le monde de l'Internet des PCs semble grandement compromise au vu des caractéristiques des objets chargés d'héberger ce service. Il faut donc repenser la façon dont celui-ci est construit [57].

2.3.3 6lowPAN

Lorsque toute nouvelle technologie apparaît, se pose la question de la réutilisation d'outils déjà connus et maîtrisés, ou la construction ex-nihilo de ce qui va permettre de manipuler cette technologie. "*Du passé faisons table rase*" offre l'intérêt d'un regard neuf, d'une adéquation directe aux besoins, mais pose le problème de l'interaction avec les technologies existantes, répandues, validées et parfois difficilement compatibles (sans parler d'une nécessaire inter-operabilité, qui va plus loin que la simple compatibilité). Par contre, la ré-adaptation d'un existant peut parfois entraîner des tels aménagements ou des distorsions par rapport à l'original, que les gains de l'interopérabilité disparaissent sous les lourdeurs des transformations et des traitements qui y sont associés. Les réseaux de capteurs seront-ils les premiers à ouvrir l'ère du post-IP [107], ou au contraire s'intégreront-ils dans le monde IP pour étendre l'Internet aux objets ?

Porter IP sur les capteurs semble a priori une mauvaise idée. En ne s'en tenant qu'au simple problème de l'adressage, l'extension d'IP aux milliards de capteurs (et autres objets) qui seront bientôt accessibles via le réseau semble intenable en terme d'étendue d'adresses disponibles. Seul IPv6 pourrait convenir en terme de puissance d'adressage, et dans ce cas, la longueur de l'entête IPv6 (à cause notamment des 128 bits d'adresse) est rédhibitoire, compte tenu de la faible taille des trames 802.15.4 (127 octets). Pourtant, cette orientation est de plus en plus largement acceptée [90]. La correspondance entre les projections des besoins en termes de nombre de nœuds et la puissance de l'adressage IPv6 plaide pour ce choix, d'autant que la technologie

4. Protocole liaison particulièrement adapté WSAN, à faible consommation, mais à faible débit.

IPv6 est de plus en plus connue, se répand [5], et n'entraîne pas de rupture importante par rapport à l'existant [141]. Mais la charge utile alors transportée par chaque paquet serait si réduite qu'elle interdirait l'utilisation de services nécessitant des messages un peu plus conséquents, disqualifiant un peu plus SOAP de la liste des protocoles envisageables.

L'apport de 6lowPAN [118] consiste en une adaptation d'IPv6 pour le protocole 802.15.4, en usage sur les capteurs [56]. La motivation des concepteurs trouve sa source dans la volonté de permettre un échange direct entre l'Internet et les capteurs, afin de se passer des passerelles applicatives nécessaires à l'interconnexion entre les services spécifiques de ces outils (de type ZigBee [12] par exemple) et le reste du monde (en IP). Si les capteurs sont capables de comprendre IPv6, alors les communications avec tous les outils existants seront simplifiées. La puissance d'adressage d'IPv6 est capable d'absorber le nombre impressionnant d'objets amenés à se connecter, tout en offrant plusieurs mécanismes particulièrement adaptés au monde des capteurs. L'auto-configuration d'IPv6 par exemple, va permettre au coordinateur du réseau 802.15.4 [14] la construction d'une adresse compatible avec son environnement. Tout en gardant le principe d'adressage fourni par ce protocole réseau, il maintiendra une table de correspondance entre l'adresse de chaque nœud (très courte) et l'adresse longue IPv6 vue de l'extérieur. Aussi, le lien direct de bout en bout entre le réseau de capteurs et les autres types de réseaux est-il établi, sans que l'action d'adaptation de la passerelle d'accès ne pénalise les échanges de façon notable.

L'utilisation d'un même et unique protocole de bout en bout nous parait préférable à celui d'une solution spécifique (tel que ZigBee par exemple) qui, pour efficace qu'elle soit grâce à son parfait ajustement au milieu, sera pénalisée lors du processus d'adaptation avec l'extérieur, majoritairement IP. En remontant vers le niveau applicatif, cette traduction peut devenir de plus en plus complexe. Les modes opératoires des mécanismes pertinents bâtis sur ces solutions spécifiques, particulièrement performant dans un environnement aussi caractérisé que celui des WSAN, peuvent être très éloignés de ceux en usage dans les protocoles applicatifs basés sur IP. La nécessaire traduction qui s'opère au niveau de la passerelle peut devenir complexe, entraînant un coût en temps de traitement pour celle-ci, de la latence et un allongement des temps de réponse pour les objets clients.

2.3.4 Services dans un environnement de capteurs

Dans le domaine assez récent des réseaux de capteurs, la nécessité de consulter les valeurs mesurées, et ce le plus simplement possible, est un enjeu majeur pour leur utilisation. Les capteurs sont capables de mesurer certaines caractéristiques de leur environnement, et peuvent ensuite transmettre ces informations au puits, chargé de les collecter. L'inversion de cette démarche permet d'imaginer un usage dans lequel c'est l'extérieur qui va interroger le capteur afin de récupérer la mesure.

5. Le 8 juin 2012 porte le nom de *World IPv6 Launch*, date du passage officiel de grands acteurs de l'Internet (Akamai, AT&T, Google, Cisco, Bing et Yahoo) à IPv6.

Dans ce cas, considérer que le capteur offre un ou plusieurs services peut devenir une option intéressante [124]. D'autres nœuds pourront alors interroger et disposer de ces services. Cette inversion de dépendance, pour reprendre la terminologie en usage dans les architectures logicielles, offre le bénéfice de limiter les échanges. Chaque nœud n'a pas à émettre régulièrement des informations mais plutôt à répondre aux requêtes, qui seront moins fréquentes (à charge du concepteur d'arbitrer entre la fréquence des réveils pour l'écoute des requêtes clientes, et les économies d'énergie).

Plutôt que d'utiliser des applications propriétaires, certainement efficaces car conçues spécifiquement pour ce besoin mais difficilement intégrables entre elles, il semble plus pertinent de décomposer les informations captées et de les transformer en services, qui seront ensuite mis à disposition via des protocoles normalisés. Le passage à une approche services permet d'envisager une récupération plus aisée, plus efficace, et surtout ré-exploitable des données collectées [21]. L'architecture orientée services des réseaux de capteurs a ceci d'attrayant qu'elle permet de respecter l'atomicité du rôle du capteur et ses capacités de traitement restreintes, et d'imaginer que c'est l'application consommatrice qui agrégera l'ensemble des données qui sont à sa disposition [103]. Les capteurs fournissent l'information de base aux applications clientes, qui les agrègent pour leurs utilisateurs [57].

Nous avons opté pour une organisation de ce type, car l'approche services diminue l'asservissement entre éléments. Le couplage faible induit par une coopération de services facilite les évolutions de l'infrastructure matérielle, et ce, tant que les services attendus sont rendus. D'autre part, cette approche services, par sa généricité, donne à des objets la possibilité de devenir eux-mêmes clients d'autres objets.

2.3.5 CoAP

Si on opte pour une approche services dans les capteurs, il faut disposer des protocoles applicatifs idoines. Dans la suite logique de 6lowPAN, CoAP [119] (*Constrained Application Protocol*) définit des méthodes permettant d'implémenter un service REST via du HTTP over UDP. Le brouillon du protocole CoAP est en cours de standardisation par l'IETF, et devrait rapidement devenir une RFC (procédure lancée en juillet 2013). En gardant toujours, comme caractéristique principale, la volonté de diminuer autant que possible les tailles des entêtes et des comportements se rapprochant le plus possible du monde IP, CoAP concentre plusieurs principes :

- Les ports et verbes HTTP sont compressés en un code défini au préalable, par la norme.
- La passerelle chargée de la communication avec l'extérieur se réfère à ce code court pour offrir l'illusion du comportement habituel du protocole réel.
- D'autre part, et toujours afin d'optimiser les échanges, et en l'absence du mode connecté TCP (trop gourmand en mémoire), des messages d'acquittement habituellement gérés au niveau 4 seront déplacés dans la couche 7 (*piggy backing*).

Des implémentations de CoAP pour PCs sont déjà disponibles : coapy [2], jcoap [6], libcoap [1], et un module pour Firefox [3]. Contiki-OS [55], le système

d'exploitation libre pour WSAN, implémente le protocole. Avec CoAP, les interactions entre services web de l'Internet des PC et de l'Internet des objets deviennent bien plus simples à réaliser, une passerelle applicative assez légère (correspondance entre les commandes REST et CoAP) se charge de l'adaptation d'un monde à l'autre.

2.4 Quelles solutions architecturales pour l'IoT ?

La création d'applications clé en main se heurte à la grande diversité des parties prenantes de l'Internet des objets, à laquelle s'ajoute le poids de l'existant. Les protocoles applicatifs de l'Internet ne sont pas adaptés, historiquement, à la puissance très limitée des objets (du moins la majorité d'entre eux, dans le cas des capteurs). Pourtant, l'Internet des objets ne peut émerger qu'à travers la nécessaire universalité de sa proposition architecturale [124], puisque la population d'objets prévue devrait dépasser de plusieurs facteurs l'actuel nombre de machines connectées au réseau des réseaux [46]. Ce qui suit est un panorama des diverses solutions envisageables pour l'intégration de tous les éléments au sein d'une structure commune, capable d'être supportée par tous types d'objets.

2.4.1 Les systèmes d'exploitation spécialisés

Plusieurs OS ont été écrits pour gérer des capteurs, notamment TinyOS [88] et Contiki-OS [55]. Le premier utilise un langage spécifique, Nes-C. Le système est organisé sous forme de modules, et ceux-ci sont appelés par l'OS via des call-backs. Le second, Contiky-OS [55], est lui aussi adapté aux capteurs et autres composants contraints. Sa principale caractéristique est le support de 6LowPAN. T. Reusing propose une comparaison des deux systèmes [110] et conclut que TinyOS peut être plus économe en énergie et mémoire sans être aussi flexible que Contiki.

Même si ces systèmes sont ouverts, et facilitent le développement d'applications destinées aux capteurs, ils ont pour inconvénient majeur la nécessité d'un investissement important du programmeur pour l'acquisition de l'expertise nécessaire à la création de programmes [44]. L'écriture d'applications pour ces systèmes requiert une durée conséquente et de la rigueur afin de respecter les limites imposées par le matériel. De plus, l'installation et la mise à jour du code généré suppose un accès physique au nœud, ce qui peut être rédhibitoire pour un utilisateur. Or, l'intégration des WSANs dans l'Internet des objets engendre des besoins plus fréquents de reprogrammation des noeuds [130], et induit des comportements plus dynamiques. Contrairement au WSAN, le rôle des objets impliqués dans une application IoT est sujet à des évolutions dans le temps [124]. Des solutions de mise à jour à distance existent (OAP, *Over the Air Protocol*), telles que (DELUGE [48] et [72], SYNAPSE [112] ou Dynamic TinyOS [98]). Cependant, elles sont généralement liées à un système d'exploitation, ce qui en limite l'usage dans un environnement fortement hétérogène tel que celui de l'Internet des objets, et ne sont pas toujours optimisées (c'est bien souvent la totalité du code qui est téléchargée à nouveau). Utiliser ce type de solutions dans notre domaine impose de limiter la diversité de la plateforme de

destination, alors que l'IoT est par nature une mise en commun de plusieurs univers, dynamiques, changeants, sur une large gamme d'appareils communicants et dont le parc est susceptible d'évoluer.

Développer une solution pour un OS précis est trop limitant au regard de la diversité du matériel impliqué. Cette restriction est l'un des principaux écueils, à notre sens, à l'essor de l'Internet de objets. Nous préférons nous orienter vers d'autres solutions qui fournissent une vision plus uniforme des objets, propice à l'évolution du parc mis en œuvre, ouvert à une large gamme de matériel.

2.4.2 Les machines virtuelles

Les machines virtuelles sont des solutions intéressantes de contournement de l'hétérogénéité. Elles masquent les différences de matériel en offrant un profil unifié et une interface normalisée. Ensuite, dans certains cas comme Java, les programmes créés peuvent prendre la forme de byte code dont la taille est souvent limitée, favorisant alors le déploiement par le réseau (légèreté et universalité).

Maté [87] est un exemple de machine virtuelle pour capteurs qui fonctionne sur TinyOS. L'utilisateur peut écrire des programmes dans des cellules de 24 instructions qui seront diffusées sur le réseau. Par contre, le langage n'est pas d'un apprentissage aisé car il ressemble à l'assembleur. De plus, Maté est limité à un seul système d'exploitation (TinyOS).

D'autres solutions sont moins contraignantes. SOS [73] par exemple, utilise un noyau commun à un ensemble de matériels, et fournit le chargement dynamique des actions à réaliser sous forme de modules. Cette solution permet une plus grande adaptabilité au matériel. Cependant, elle impose la connaissance du langage C et une compréhension assez technique du comportement du système.

Un autre portage intéressant est celui de la machine virtuelle Darjeeling [38], proche de Java, et adaptée à des architectures matérielles de type capteurs avec des processeurs 8 ou 16 bits et offrant de 2 à 10 kilo-octets de mémoire vive. Le résultat est fonctionnel, et permet l'exécution de programmes WORE (écrits une fois, exécutables partout). Cette universalité est propice à l'écriture d'applications Internet des objets. Toutefois, les besoins applicatifs sur chaque nœud devraient pouvoir être décrits au moyen de langages plus simples et adaptés que Java. Darjeeling n'offre qu'un sous-ensemble des fonctionnalités du langage, impliquant une connaissance précise du programmeur sur les limites de la machine virtuelle par rapport à la norme du langage.

2.4.3 Vers une architecture orientée services

Suivant le même raisonnement que pour l'Internet des PCs, nous regroupons ici les propositions d'exploitation des ressources offertes par les capteurs en les présentant sous forme de services. D.B. Green présente les atouts de cette approche dans [67]. Puisque la puissance d'adressage d'IPv6 permet de gérer le grand nombre de capteurs susceptibles de se connecter, un lien de type Machine-to-Machine (M2M)

utilisant les protocoles connus de la SOA peut être construit grâce à des outils de "*mashup*" [6]. Le temps nécessaire au développement d'une interaction entre objets devrait être fortement réduit par rapport à la programmation de la même application avec les outils habituels fournis avec les objets. Ces concepts sont mis en œuvre dans une implémentation pour TinyOS décrite dans [111] par A. Rezgui et al. dans leur solution TinySOA. De la même façon, C. Buckl et al. font le choix d'offrir directement les services sur le nœud lui-même plutôt que de figer ce rôle sur la passerelle d'accès [39]. Aussi les accès seront-ils orientés services de bout-en-bout. E. Wilde décrit aussi un "Web des objets" [135] dans lequel tout, des capteurs aux serveurs les plus puissants, dialogue en utilisant directement les mécanismes standards du Web.

L'intérêt de cette solution tient au fait qu'elle offre le couplage faible nécessaire à l'indépendance entre les différents acteurs de l'application. Outre sa concordance avec notre vision de l'Internet des objets qui prône les interactions entre éléments, elle libère le programmeur d'une forte dépendance au matériel, celui-ci étant perçu par le filtre des services qu'il rend. Aussi les évolutions matérielles ou les remplacements par des substituts aux propriétés identiques deviennent plus aisés. Le problème résiduel est de pouvoir facilement définir, et mettre à jour, ces services.

2.4.4 Les Framework

Un framework (en français cadriciel) est un "patron" d'application pré-organisé. L'infrastructure est fournie, et l'utilisateur peut commencer à décrire sa logique métier. Tout le cadre nécessaire au bon fonctionnement est fourni, seul la partie spécifique doit être développée.

SENSEI [104] [126] est une solution de ce type. Développé dans le cadre du projet européen FP7 (septième programme cadre européen pour la recherche et le développement technologique), ce framework très complet, orienté capteurs et effecteurs, contient des outils de découverte des nœuds, de stockage de ressources (un nœud détecté devient une ressource utilisable), un outil de requête et un analyseur sémantique chargé de la compréhension des demandes des utilisateurs. Les principaux protocoles sont gérés (ZigBee, 6LowPAN, IP). L'outil est cependant orienté données et requêtes, et sert surtout à déclencher des interactions entre l'utilisateur et les objets. Il lui manque une dimension Machine-to-Machine (M2M).

SOA4ALL [53] est un projet basé sur un framework et une infrastructure dont l'objectif est de mettre à disposition de l'utilisateur tout un ensemble de ressources de l'Internet qui apparaissent alors sous forme de services. L'architecture décrite dans [86] comprend un moteur de composition chargé de lier les services et un analyseur sémantique interprétant la demande de l'utilisateur. L'ensemble est cependant centralisé, organisé sous forme d'*orchestration* (ce qui ne nous semble pas optimal), et l'univers des capteurs et leurs spécificités ne sont pour le moment pas pris en compte dans la solution.

6. Solution graphique pour composer des applications à partir d'éléments que l'utilisateur relie entre eux.

V. Foteinos et al. proposent un framework spécialisé dans l'IoT [63] auquel ils ajoutent une dimension sémantique plus poussée, notamment en employant un moteur d'inférence chargé d'adapter les ressources selon le contexte, la fiabilité et la disponibilité des éléments, et d'en masquer l'hétérogénéité. Leur solution manipule des objets virtuels correspondant aux besoins de l'utilisateur, et tente dans les couches plus basses de résoudre cette virtualisation en recherchant les objets réels capables d'y répondre. L'ensemble fournit une approche de haut niveau, mais pour le moment principalement en direction des utilisateurs, sans offrir une expressivité pour les interactions d'objets entre eux.

Enfin, FI-Ware [16] est un projet européen financé par le FP7 et des partenaires privés européens (industriels de l'informatique, des télécommunications ou de l'électronique) dont le but est de fournir le framework au cœur d'une plate-forme commune pour l'Internet des objets. Ce framework est constitué d'un entrepôt de service (*store*), d'un analyseur sémantique, et permet à des fournisseurs d'offrir des services, et aux utilisateurs de les consommer et de les déployer sur leurs objets. L'ensemble a une orientation clairement *business*.

Ces solutions nous semblent particulièrement indiquées pour l'Internet des objets. Cependant, nous voulons nous orienter vers des solutions chorégraphiées, mettant en jeu des objets aux comportements autonomes. D'autre part, nous voulons pouvoir piloter les objets à distance et définir ces comportements à la volée, ce qu'aucun des projets présentés ci-dessus n'est capable d'offrir.

2.4.5 Les middlewares

Un middleware (ou *intergiciel*) est une solution complète chargée de s'interfacer entre deux éléments, une application d'un coté et, de l'autre, tout un assortiment allant du matériel au logiciel, en passant par des solutions de persistance. Le Middleware est chargé d'adapter, de façon logicielle, un monde à un autre. Ces solutions de médiation sont encouragées par H. Sundmaeker et al. [124] et apparaissent par exemple dans [31] [25]. Les middlewares pour les WSAN ont fait l'objet d'une étude comparative [75].

Dans le monde des capteurs, SensorWare [35] tient à la fois du middleware et de la machine virtuelle. SensorWare permet le déploiement à la volée de scripts à exécuter sur les nœuds, et l'accès aux services offerts par ces scripts. La dynamicité de la solution permet au programmeur d'être très réactif. Cependant, SensorWare souffre d'une implémentation très lourde en terme de capacité mémoire, 4 fois supérieure à la taille offerte par un TelosB par exemple, ce qui en prohibe l'usage.

La proposition Octopus [26] de F.J. Ballesteros et al. est construite à partir d'une vision un peu particulière du middleware que les auteurs définissent sous le terme d'*Upperware*. A contrario du middleware qui implique l'appel explicite à ses fonctions, le rendant intrusif, la notion d'Upperware est complètement transparente. Si Octopus fonctionne sur la machine hôte, alors l'accès à ses services s'effectue via le gestionnaire du système de fichiers. S'il n'est pas présent, les fichiers ne sont pas accessibles (à l'image du pseudo répertoire */proc* de Linux). Ce dispositif permet

les communications entre les nœuds de façon assez naturelle et souple. Cependant, reste ici à régler le problème de l'algorithme à exécuter sur le nœud. Il faut donc être capable de décrire la logique, avec un langage supporté par la plateforme d'accueil. L'utilisateur se retrouve à nouveau confronté à des problèmes d'expertise en langage de programmation, et leur multiplicité induite par la grande diversité des matériels à l'œuvre.

2.4.6 Les enjeux de l'IoT

L'Internet des objets est à la croisée de plusieurs domaines et suscite l'intérêt de la communauté académique comme des industriels. Il s'agit de ne pas passer à coté des opportunités associées à l'essor de l'IoT.

A cet égard, les travaux de F. Pinto [42] sont intéressants à deux points de vue. D'un coté, il présente l'Internet des objets d'un point de vue plutôt orienté Machine-to-Machine, ce qui constitue l'angle par lequel nous avons choisi d'aborder le domaine. Les techniques que nous avons présentées dans ce chapitre ne permettent pas de construire des automatismes universels d'interactions logiques entre les objets. Ce sont ces actions et réactions de notre environnement qui nous semblent importantes à concrétiser, à automatiser, à décrire, et ce de la manière la plus simple possible, en offrant un certain niveau de sémantique dans l'échange. Aussi partageons-nous le point de vue de l'auteur, et notre objectif consiste à résoudre le problème selon cet angle d'approche.

Le second point de vue de l'auteur concerne son approche business. Il propose des pistes afin que les fournisseurs d'accès Internet et des services associés bâtissent leur modèle économique, alors que certains d'entre eux se montrent circonspects envers le domaine, y voyant une charge en terme de trafic sans contrepartie génératrice de valeur ajoutée (cf *Industry & Business panel ICC 2013*[7]). Ce point de vue montre que l'industrie elle aussi s'intéresse au domaine. L'Union Européenne est de son coté active, notamment par les financement de ses projets (*FP7 : 7ième programme cadre*) et présente dans de nombreux programme de recherche, aussi bien au niveau pratique (comme le Future Internet F-I) qu'au niveau architectural (voir par exemple IoT-6 [141] qui liste un ensemble de règles et de bonnes pratiques à respecter pour résoudre les problèmes de construction d'une architecture valide pour l'IoT).

Les enjeux du domaine dépassent largement celle d'une nouvelle technologie. L'Internet des objets pose les jalons de développements futurs puisque les solutions qu'il nécessite auront des impacts en dehors de ses frontières. La réalisation d'une communication généralisée et universelle entre toute une gamme d'outils donne accès à un niveau plus fin de la connaissance du milieu, et ouvre la voie à des approches plus cognitives [63]. Permettre aux objets d'interagir en autarcie est un pas vers l'informatique pervasive. D'autres domaines, par ricochet, bénéficieront des apports des solutions de l'Internet des objets. Ainsi, les notions autonomiques [64] et [113] dans lesquelles des appareils actifs du réseau sont capables de s'auto-configurer,

7. Table ronde dans laquelle des opérateurs ont exprimé leur réserve concernant l'augmentation de la consommation de la bande passante, sans l'ajout de valeur correspondante.

auto-optimiser, auto-protéger et s'auto-réparer pourraient trouver dans les fondations de l'IoT les outils nécessaires pour interagir et construire des ilots de gestion distribués [91] [45].

2.4.7 Les solutions retenues pour notre proposition

Afin d'obtenir la meilleure interconnexion possible entre les différents réseaux qui composent l'IoT, et permettre aux différents éléments de communiquer, nous avons préféré nous orienter vers les solutions qui nécessitent le moins d'intermédiaires chargés de l'adaptation des flux, et sélectionné celles qui offrent le lien le plus direct de bout en bout. Dès l'origine de nos travaux, le protocole IPv6 nous a semblé le meilleur choix capable de supporter le nombre important d'appareils mis en œuvre sur l'Internet des objets. Différents travaux ont montré que, malgré la charge représentée par ce protocole au regard des faibles capacités des capteurs, son adaptation aux objets est tout à fait possible. A. Dunkels en a fait la démonstration [56] tout d'abord, puis la RFC 4919 [85] est venue valider 6LowPAN [118].

Notre travail, au démarrage de cette thèse, a commencé par le pressentiment que la disponibilité de ce protocole aurait pour conséquence une simplification de l'accès aux objets, et autoriserait alors leur intégration rapide et simple aux réseaux existants. Ce choix semble être validé par de récents travaux sur IPv6 et les capteurs [140] [131]. Enfin, s'appuyant sur 6LowPAN, le protocole CoAP [119], fruit des travaux de Z. Shelby, propose une compatibilité REST aux objets. CoAP nous a fourni la base sur laquelle l'ensemble de notre solution est construite. Ce protocole, qui n'était qu'une hypothèse de travail au moment de la conception de notre solution, vient d'être proposé en tant que standard par l'IETF au moment où nous écrivons ces lignes (15 juillet 2013).

Le domaine génère d'ailleurs une grande activité à la fois au niveau de la recherche et de la part des industriels. Les problèmes d'architectures constituent la suite logique à traiter et à normaliser (voir les prémisses du groupe de travail sur une architecture spécifiquement adaptée à l'Internet des objets [24]).

L'intégration des réseaux d'objets à l'Internet conduit à appréhender cet ensemble hétéroclite dans sa globalité, en proposant une solution à la fois respectueuse des limites de chacun, et capable d'intégrer des constituants les plus variés. Une architecture, qu'elle soit pour l'IoT ou pour tout autre domaine, doit définir les différentes couches, leurs comportements et caractéristiques, l'organisation de l'ensemble, ainsi que la formalisation des échanges à chaque niveau. Notre point de vue nous oriente vers une organisation distribuée dans laquelle chaque objet, autonome, réagit aux stimuli et à sa perception de l'environnement. Ainsi, la chorégraphie des services offerts, l'abstraction uniformisant la représentation des objets, et la contractualisation des échanges sont les principales notions qui guideront notre réflexion.

2.4.8 L'intégration des objets dans l'Internet

Nous retenons de l'ensemble des travaux présentés ci-dessus certaines démarches qui nous semblent correspondre aux pistes à suivre afin d'évoluer dans le cadre dans lequel nous nous sommes placés. Nous avons choisi de considérer l'Internet des objets comme la continuité, l'extension de l'Internet des PC. Dans cette vision, les objets proposent alors eux-mêmes des services que les acteurs habituels (utilisateurs, autres ordinateurs) vont consommer et avec lesquels ils vont interagir.

Il s'agit donc de reproduire sur ces objets hétéroclites un comportement le plus proche possible de celui attendu d'un serveur. Les objets devront être programmables, afin de livrer un service évolutif et adapté. Les premiers travaux ayant retenus notre attention sont ceux de S. Duquennoy. Sa thèse [58] présente des solutions pour l'Internet des objets bâties sur des serveurs web hébergés sur chaque objet, baptisés Smews. Cette implémentation valide le concept, et utilise des méthodes de pré-calcul des paquets permettant de composer ceux-ci à la volée, avec du contenu dynamique, tout en allégeant le traitement nécessaire à la fabrication des messages, le tout pour une empreinte mémoire réduite.

Les travaux [69] de D. Guinard conduisent également vers une implémentation de services web directement sur les objets impliqués dans une application IoT. Cette démarche organise un Internet des objets cohérent de bout en bout, sans passerelle ni adaptation/transformation d'un monde à un autre. Plusieurs autres travaux [120][70] de cet auteur prônent cette démarche SOA pour l'IoT.

Enfin, M. Kovatsch, dans ses récents travaux [83], propose un serveur très léger, programmable, interagissant avec les objets, le *thin server*. Ce serveur, ou du moins chacune des instances représentant l'objet distant, peut ensuite être rapidement reprogrammé par l'utilisateur via à un moteur de composition, Actinium [82]. Actinium recense tout un ensemble de comportements que les objets peuvent adopter, et en permet l'activation et l'interaction.

Notre démarche trouve des similitudes avec celles présentées ci-dessus. Elle diffère par une orientation plus distribuée, et la mise en œuvre d'une virtualisation de chaque objet, autorisant une portabilité accrue des comportements déployés. Nous montrerons dans le chapitre 3 comment la distribution de la logique allège les effets de l'application complète sur les réseaux contraints, tandis que la couche d'abstraction matérielle, proposée dans le chapitre 4, favorise l'intégration de nouveaux composants. Nous nous attacherons tout au long de cette thèse à montrer les atouts d'une chorégraphie des services offerts par les objets. Enfin, afin de contrôler la validité des échanges entre chacun d'eux, au regard des attentes et productions des services qu'ils hébergent, la modélisation et les méthodes de validation introduites au chapitre 5 garantiront la conformité de la composition de l'utilisateur.

2.5 Conclusion

Alors que l'Internet des objets suscite des intérêts toujours plus nombreux, sa définition même recouvre encore différentes acceptations. La convergence de tous les

outils capables de traiter de l'information et de l'échanger avec d'autres (du plus petit capteur aux plus puissants effecteurs, des smartphones jusqu'aux serveurs clouds, des services des réseaux sociaux en passant par le mobilier urbain, des dispositifs présents dans les immeubles intelligents aux véhicules et équipement d'aide à la circulation, etc.) conduit à imaginer de usages divers et variés. Ces vues multiples de l'informatique pervasive et ubiquitaire, proposées sous le terme "*Internet des Objet*", ne sont cependant pas contradictoires. Elles impactent à la fois les organisations proposées, les contraintes à résoudre, et les usages attendus.

Selon le point de vue de l'observateur, et le postulat de départ, les différentes approches justifient la multitude de pistes et d'études, aux conclusions parfois dissemblables, dont la littérature recèle. Cet ensemble hétéroclite, à l'image du domaine qu'il tente de cerner, ne doit pas surprendre par ses apparentes contradictions. L'Internet des objets est un vaste sujet dont les délimitations restent à définir, et dont l'utilisation demeure encore à ses balbutiements.

Proposer une architecture, résoudre les problèmes rencontrés aux différents niveaux afin de réaliser une architecture fonctionnelle pour l'IoT, quantifier les impacts de l'organisation proposée sur les réseaux d'objets nouvellement intégrés, voilà l'objectif des chapitres suivants qui détaillent notre proposition de solution complète, procurant à l'utilisateur une expérience concrète d'interactions entre ses objets, par la construction dynamique, intuitive et évolutive, d'applications Internet des objets.

Chapitre 3

Impacts des Architectures Orientées Services sur les réseaux de capteurs

Sommaire

> Il fallait un calculateur, ce fut un danseur
> qui l'obtint.
>
> Les noces de Figaro
> Pierre Augustin Caron de Beaumarchais

Omniprésent et ubiquitaire, tel est, en 1991, "l'ordinateur du XXI^{e} siècle" pressenti par M. Weiser [133]. Dans cet article fondateur de l'Internet des objets, l'auteur prédit une évolution du "*many-to-one*" (de nombreux utilisateurs partageant une ressource unique) vers le "*one-to-one*" (l'ordinateur personnel), puis "*one-to-many*" (un seul utilisateur dispose de multiples machines). En 2011, D. Evans [60] estime que le "*one-to-one*" a été atteint entre 2008 et 2009, ouvrant l'ère de l'Internet des objets. L'auteur projette le nombre de 50 milliards d'appareils connectés pour 7 milliards d'êtres humains d'ici 2020.

L'établissement de la nécessaire interaction entre tous ces appareils peut s'appréhender selon deux grandes méthodes : soit une forme *centralisée* dans laquelle chaque élément est sous le contrôle d'un point central, ou bien *distribuée* sur les différentes parties prenantes, chacune interagissant avec ses partenaires selon ses besoins. Ces deux organisations se retrouvent aussi bien dans l'architecture orientée services (SOA pour *Services Oriented Architecture*) présente sur l'Internet [102] que dans les réseaux de capteurs avec ses modes *automatisés* et *semi-automatisés* [19]. Le choix de l'une ou l'autre des solutions engendre des trafics différents sur les réseaux traversés. Nous nous sommes intéressés ici à l'observation de ces deux modes, principalement en termes de longueur du chemin parcouru par les données échangées. Les caractéristiques des différents réseaux mis en œuvre dans l'Internet des objets étant très variées, les résultats de notre étude aident dans la décision d'opter pour l'une ou l'autre des solutions, au regard des atouts et contraintes du réseau utilisé.

Ce chapitre s'intéresse principalement à l'impact des applications centralisées ou distribuées sur la longueur des parcours des messages à l'intérieur d'un réseau. Après avoir les avoir décrits, nous montrerons les similitudes entre les deux modes, *automatisé* et *semi-automatisé*, des WSAN et ceux de l'architecture SOA (*orchestration* et *chorégraphie*). Puis nous analyserons théoriquement et expérimentalement les effets de ces architectures logicielles, et présenterons une quantification comparative et les bornes en termes de gains maximaux générés. Enfin, nous vérifierons sur banc de test la conformité de la prédiction du comportement en observant les impacts d'une application réelle.

3.1 WSAN automatisés et semi-automatisés

L'Internet des objets consiste à intégrer de multiples réseaux, et les services et matériels qui les composent, et ce grâce aux protocoles fédérateurs de l'Internet [31] et plus spécifiquement du Web (on parle alors de *Web des objets*) [124]. Il s'agit notamment de faire communiquer les réseaux de capteurs/effecteurs (*Wireless Sen-*

sors and Actuators Networks ou WSAN), les réseaux PAN (*Personal Area Network*) avec le réseau des réseaux [90]. Cette extension de l'Internet, que nous définissons sous le terme d'*Internet capillaire* [1], va favoriser l'interaction entre les appareils portés par les utilisateurs (smartphones, appareils voire même vétements disposant de RFID, bluetooth, etc.), leur environnement (réseau de capteurs dans l'habitat, le lieu public, la ville, etc.) et les services habituellement accessibles de l'Internet. Cette interconnexion peut s'effectuer principalement de deux façons [125] :

soit en trouvant un mode d'échange commun à toutes les parties ;
soit en utilisant des passerelles adaptant chaque monde aux autres.

A l'intérieur de ces réseaux, les nœuds s'organisent pour l'accessibilité de l'ensemble des informations offertes. Concernant les réseaux de capteurs, l'infrastructure du réseau est informelle, les nœuds situés à des positions aléatoires, non prévisibles et éventuellement changeantes. Les capteurs doivent s'organiser en utilisant leur capacité à s'échanger de l'information entre eux, avec les autres nœuds à portée. Ils doivent ensuite se structurer en construisant des topologies variées, hiérarchiques ou maillées [19] afin d'atteindre les éléments plus distants.

De par leur participation majeure (bien que non exclusive) à l'Internet des objets, les WSAN et leurs contraintes doivent être pris en considération.

3.1.1 Les WSAN et leurs contraintes

Les réseaux de capteurs sont constitués d'éléments équipés de capacités de mesure du monde environnant (*capteurs*) ou de moyens d'action sur cet environnement (*effecteurs*) [19]. Ces appareils, emblématiques du terme *objets* (sans toutefois que celui-ci soit restreint aux seuls capteurs et effecteurs), sont dotés d'un processeur, de mémoire, et d'un accès au réseau sans fil. Conçus pour être autonomes, ils disposent pour ce faire de leur propre réserve d'énergie. En outre, leur taille doit être des plus réduites afin de s'intégrer au mieux dans l'environnement. Toutes ces caractéristiques engendrent une multitude de contraintes : la puissance du processeur est faible, la capacité mémoire restreinte, le débit du réseau est limité et la taille des paquets gérés réduite [136]. Enfin, la contrainte majeure de ces outils concerne leur consommation en énergie [23]. La batterie n'est pas renouvelable et l'objet doit pouvoir fonctionner le plus longtemps possible sans intervention humaine. L'utilisation des ressources du nœud doit donc être parcimonieuse.

Cette limitation très forte en énergie conjuguée avec l'impératif de longévité plaide pour une manipulation des informations en local puisque cela est non seulement possible mais moins coûteux. Ce traitement localisé des données est préféré à leur transfert [18], sous réserve que l'algorithme de l'application se prête au traitement local des informations et que cela ait du sens.

1. Puisqu'il s'étend plus profondément dans le monde réel, en liens de plus en plus fins, à faible débit, à l'image des vaisseaux sanguins capillaires.

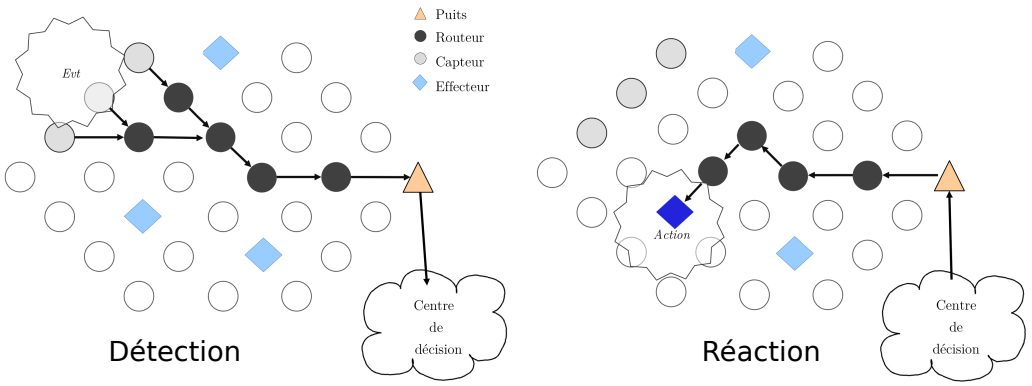

FIGURE 3.1 Dans un réseau de capteurs et d'effecteurs en mode *semi-automatisé*, les données captées remontent au *puits* afin d'être traitées, et l'ordre redescend vers l'effecteur. Les nœuds proches du *puits* sont sollicités.

3.1.2 Les organisations automatisées ou semi-automatisée

Historiquement, les réseaux *Wireless Sensors Network* (WSN) ont d'abord été constitués de capteurs uniquement, et servent donc à obtenir des mesures de l'environnement. Ces valeurs sont transmises vers le puits (le *sink*) pour analyse par des appareils plus puissants, non limités en énergie ni en puissance de traitement. *"Les données sont envoyées au puits, ou le puits demande la valeur de la mesure physique"* [132]. Le réseau de capteurs doit s'auto-organiser pour la transmission des données, le plus souvent en utilisant le principe de multi-sauts pour atteindre la destination (il est préférable de transmettre à un voisin qui jouera le rôle de relais plutôt que d'essayer d'émettre jusqu'au *puits*, opération trop consommatrice en énergie ou tout simplement irréalisable si le *puits* est hors de portée).

L'arrivée des effecteurs [136], formant alors les WSAN (*Wireless Sensors and Actuators Network*) a quelque peu modifié l'organisation de ces réseaux. Dans ce cas, les informations de pilotage ou des ordres sont susceptibles de parvenir à l'effecteur en parcourant le réseau dans le sens inverse de celui utilisé jusqu'alors. Pour peu que ces ordres découlent de décisions prises à la suite de mesures effectuées par les capteurs environnants, le raisonnement qui consiste à faire remonter les données vers le *puits* puis faire redescendre les ordres vers l'effecteur semble moins pertinent au regard notamment des économies d'énergie requises. La mesure, le traitement et la prise de décision directe entre capteurs et l'effecteur mobilisent un circuit plus court et autorise une utilisation optimale du réseau [19]. Ce mode dit "*automatisé*" (Y. Akyildiz évoque *"une interaction directe [..] aussi appelée "coordination"*) s'oppose au mode "*semi-automatisé*", dans lequel les données effectuent des aller-retours vers le *puits* (et par delà, au centre de décision).

Les figures 3.1 et 3.2 décrivent les différences de traitement d'un même événement capté et géré selon un WSAN organisé en mode "*semi-automatisé*" ou "*automatisé*". Dans le premier cas, les capteurs qui détectent l'événement communiquent avec le *puits*. L'information parcourt une partie importante du réseau afin d'être traitée en

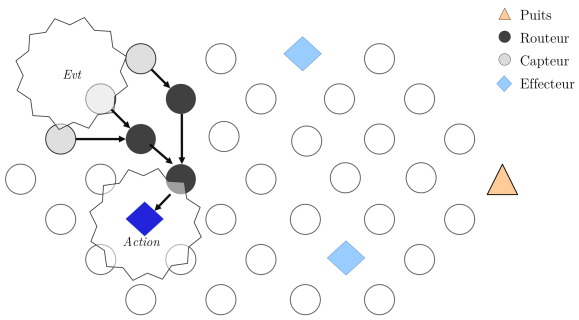

FIGURE 3.2 Le mode automatisé en application dans un WSAN : si l'effecteur chargé de réagir est proche du capteur détectant l'événement, l'utilisation du réseau est optimisée.

dehors de celui-ci, le *puits* faisant office de passerelle vers l'extérieur. Si la conclusion du traitement conduit à agir sur un effecteur situé à l'intérieur du même réseau, l'ordre correspondant redescendra via le *puits* vers le nœud visé. Il est plausible que l'effecteur chargé de la réaction face à l'événement soit assez proche du capteur [19], d'où la prépondérance de ce cas de figure.

La remontée de l'information au *puits* entraîne la mise en œuvre d'un nombre plus important d'éléments du réseau par rapport à un traitement local tel que présenté dans la solution du mode "*automatisée*" (figure 3.2). Dans ce cadre, le capteur, par sa connaissance du réseau et ses capacités d'analyse des mesures effectuées, détermine l'action à déclencher sur l'effecteur. Il peut donc directement se mettre en rapport avec cet effecteur, raccourcissant alors le cheminement du message. Pour l'obtention du même résultat, le mode "*automatisé*" paraît intuitivement faire un usage plus pertinent et allégé du WSAN. De plus, dans le cas du mode *semi-automatisé*, on note que la surcharge en termes de communications n'est pas uniformément répartie, mais affecte plus particulièrement les nœuds à proximité du *puits* [18] qui sont plus souvent sollicités.

3.1.3 Les WSAN et l'Internet des objets

Dans le cadre des WSAN, on pressent intuitivement que le mode "*automatisé*" est plus efficace [96]. Avec l'introduction des effecteurs, les messages circulent dans les deux sens, montant et descendant. Supposer une certaine proximité entre les effecteurs capables d'actions spécifiques et les capteurs mesurant les valeurs à l'origine de la décision de ces actions semble réaliste. Mais la problématique qui apparaît alors tient dans la difficulté à réaliser l'asservissement d'un appareil à un autre, puisque dans le mode *automatisé*, ce sont maintenant des traitements qui interagissent, et non plus de simples données qui sont échangées. L'adéquation entre le mode *automatisé* et l'usage parcimonieux de l'énergie fait apparaître le besoin de solutions applicatives. Il faut permettre l'invocation de l'action des effecteurs par des capteurs, et ce de la façon la plus simple possible, malgré les capacités de programmation li-

mitées de ces appareils. Enfin, l'intégration des WSAN dans le sur-ensemble de l'Internet des objets tend à en modifier l'usage [124] : ils devront être capables d'interagir avec une gamme plus large d'appareils, et de s'intégrer dans des applications de plus en plus variées.

Cette convergence plaide pour la "*conception d'applications collaboratives entre nœuds*" [27]. Construire un Internet des objets pose des problèmes d'hétérogénéité des matériels, car "*une telle diversité est à l'opposé d'une organisation distribuée du logiciel*" [22]. Usages différents, intégration avec des technologies bien connues et répandues, l'Internet des objets peut s'appréhender sous la forme d'une architecture logicielle de type SOA [124]. Cette solution facilite les interactions entre différentes unités de traitements autonomes, en cachant les traitements derrière des interfaces dont l'accès est simplifié.

Z. Shelby, qui est à l'origine d'un grand nombre d'avancées dans le domaine, a écrit qu'"*il est pertinent de voir le WSAN dans son ensemble comme une architecture orientée service (SOA)*" [117]. Les deux modes (*automatisé* et *semi-automatisé*) proposés pour l'utilisation des réseaux WSAN font écho aux méthodes d'organisation logicielles de la SOA. Les prochains paragraphes s'intéressent plus particulièrement à décrire ces architectures applicatives (*chorégraphie* et *orchestration*) répandues dans l'Internet des PC.

3.2 *SOA : chorégraphie* et *orchestration* de services

Nous allons tout d'abord définir, d'une manière globale, ce que sont les deux approches de l'architecture d'applications bâties autour de la composition de services. Puis nous nous intéresserons aux avantages et inconvénients de ces *Architectures Orientées Services* (SOA) dans leurs utilisations traditionnelles dans le cadre de l'Internet des PC.

3.2.1 L'*orchestration* : la centralisation du traitement

La première approche concernant la collaboration des services s'appuie sur un concept de centralisation du contrôle du déroulement de l'application. Héritée des méthodes de développement structuré, l'*orchestration* de services [102] aborde la problématique de l'utilisation de la puissance de traitement de machines distantes et inter-connectées en reproduisant les pratiques de ce mode de programmation, telles que les appels de fonctions, les paramètres, les valeurs de retour, etc. En l'occurrence, le programme principal, exécuté sur le nœud central, garde le contrôle complet du déroulement de l'algorithme. Il utilise les services déportés sur les serveurs avec lesquels il collabore le plus souvent en mode synchrone (l'attente de la réponse du service bloque l'exécution du programme). Il récupère alors le résultat de cet appel, l'intègre dans sa propre gestion et poursuit son déroulement (voir figure 3.3).

Dans ce type d'organisation, seul le nœud central détient le contrôle de l'application. Cela n'empêche pas les services utilisés d'avoir leur propre progression logique.

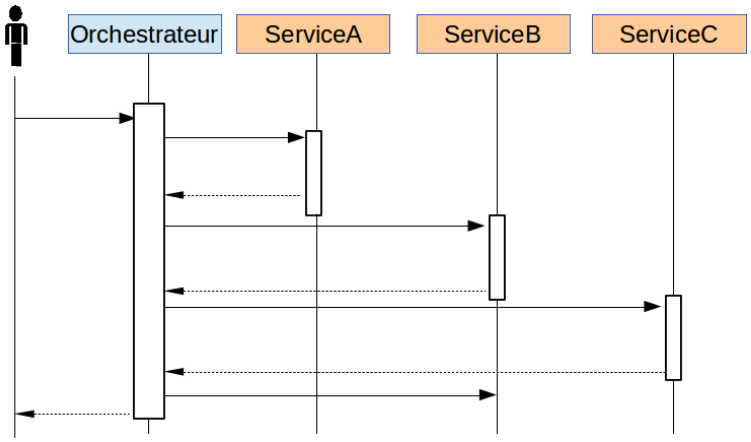

FIGURE *3.3* Diagramme de séquence d'une orchestration : Un utilisateur déclenche une action sur l'orchestrateur. Celui-ci appelle et combine les résultats fournis par les Services A, B et C. L'utilisateur n'a aucune vue sur ces différents services, le résultat provient du traitement de l'orchestrateur

Cependant, le point central domine les autres participants, car il est le seul à disposer de la vue complète de la logique applicative et de sa progression. La dépendance est clairement orientée, les services participant n'ayant pas la possibilité d'inverser le sens de la prise de décision.

L'*orchestration* de services est une technique couramment utilisée dans le domaine des applications sur l'Internet. Qu'il s'agisse de sites marchands proposant à l'internaute des biens de consommation, des solutions complètes de réservation de voyage, des réseaux sociaux et l'économie des services associés, en passant par les moteurs de recherche ou les fournisseurs de "*Cloud*" et services en ligne, on retrouve ce modèle de communication utilisant des services Web. Le site marchand propose ses produits, mais le service qu'il offre agrège celui des sociétés de distribution de colis chargé du transport. Il en va de même pour les services web des banques pour le règlement de l'achat. Les services utilisés sont des composants qui suivent leur propre logique, mais ne peuvent prendre le contrôle de l'application de vente en ligne. Les sites de vente de voyages adoptent le même comportement, mixant les résultats des web services des compagnies aériennes et des hôtels. En matière de *cloud* et autres services en ligne et réseaux sociaux, de nombreuses interactions concernant les comptes par exemple (authentification des utilisateurs), des collectes de données (statistiques sur le comportement de l'internaute) sont réalisées, toujours sous contrôle de l'application principale. La figure 3.3 montre des échanges entre différents services menés par un orchestrateur. Grâce à sa vision globale de l'application, celui-ci déclenche les appels aux différents services.

3.2.2 La *chorégraphie* : le traitement distribué

Certaines applications dépendent parfois moins directement d'un élément central. Dans certains cas, la responsabilité du déroulement de l'ensemble des réactions

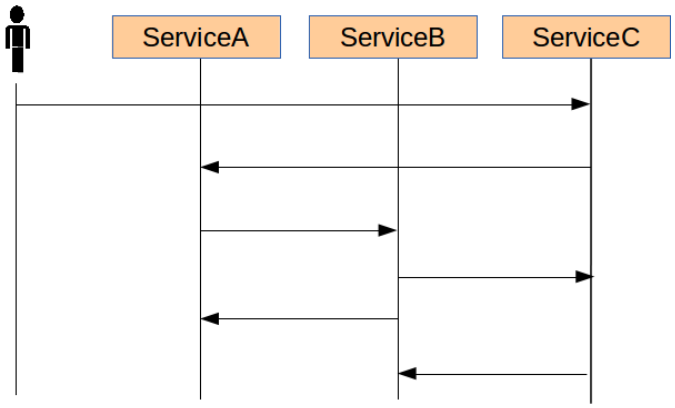

Figure 3.4 Diagramme de séquence d'une chorégraphie : L'utilisateur déclenche ici une action sur un des services. Celui-ci déclenche à son tour des appels asynchrones sur les autres services mis en jeu.

peut être répartie sur plusieurs points de contrôle, chacun pouvant prendre à son tour le rôle de décideur et piloter ainsi le déroulement des échanges. Dans ces cas de figure, la centralisation imposée par l'*orchestration* s'avère inadaptée. A l'image traditionnelle du "*chef d'orchestre*" qui dirige un ensemble se substitue celle d'une salle de bal dans laquelle chacun des danseurs (ou couple de danseurs) exécute son pas de danse suivant un schéma qui lui est propre, interagit avec les autres participants (évite les collisions et trouve sa place), le tout sans qu'aucun gestionnaire ne coordonne l'ensemble. Dans cette vision d'un contrôle *distribué* plutôt que *centralisé*, les interactions entre tous se font d'égal à égal, résultant d'actions et de réactions par rapport à la perception de l'environnement immédiat de chacun, des informations connues, et de la logique propre à chaque élément. La majorité des échanges est plus directe, les décisions prises au plus proche de l'impact de leur réalisation. Pour revenir à l'image qui a donné son nom à la *chorégraphie*, il serait bien difficile de construire un système aussi efficace pour la bonne gestion des danseurs du bal en utilisant un algorithme centralisé. Celui-ci demanderait un grand nombre de communications et le traitement de nombreuses informations pour un résultat général probablement plus aléatoire et moins fluide.

A l'image du bal, la *chorégraphie* en SOA [102] est conçue comme une forme de collaboration des parties prenantes sans que l'une d'elles ne prévale sur les autres. Là aussi, chaque participant poursuit son objectif, réagit aux messages reçus, et utilise les informations transmises par les partenaires. La difficulté réside dans la conception d'algorithmes spécifiques à chaque nœud, décrivant la tâche qui lui incombe, tout en permettant dans le même temps d'atteindre l'objectif commun (s'il en existe un). Cela est souvent moins difficile qu'il n'y paraît, puisqu'ici l'accent est mis sur la cohérence du point de vue de chaque entité partie prenante. L'idée est que chacun respecte sa propre logique de comportement, selon les informations accessibles de son propre point de vue. La somme des comportements valides permet à l'ensemble de rester dans des bornes satisfaisantes. La figure 3.4 visualise les échanges lors de

l'appel d'un service (*ServiceC*), qui entraîne l'utilisation d'autres services. Au final, le résultat équivaut celui de la figure 3.3 bien que les processus de décision soient distribués dans ce dernier exemple.

3.2.3 Les avantages et inconvénients des deux approches

Parce qu'elle est pensée dans la continuité de la programmation structurée, l'*orchestration* a été logiquement la première approche pour concevoir la composition de services. La centralisation du raisonnement et le rendu de services clairement identifiables en tant qu'entité logique atomique (comme une fonction) permettent la construction d'applications SOA. Les avantages de l'*orchestration* sont fondés sur l'unicité géographique du centre de décision, ainsi que sa mainmise complète sur le déroulement logique. Cependant, en cas de panne du moteur de composition ou de l'un des services (services synchrones bloquants), l'application est mise en péril. D'autre part, l'*orchestration* utilisée sur une architecture de communication sous forme d'arbre par exemple, engendre une surcharge des liens, notamment à cause des échanges avec le nœud central, tandis que d'autres voies sont elles très sous-exploitées (les voies transverses). Là encore, l'application sera fragilisée par la perte d'un lien vers le centre, surtout plus ce lien est proche du nœud central de décision (les nœuds dits critiques [106]). Le fait d'être sur-consommateur du lien dont il dépend introduit une faiblesse dans le système, et peut avoir des conséquences notamment si (et c'est le cas des réseaux contraints) les deux phénomènes s'opposent : la sur-exploitation d'un capteur entraîne plus rapidement sa mise hors service (énergie épuisée), obligeant alors une réorganisation du réseau afin de lui trouver un remplaçant, et ce jusqu'à ce qu'aucun nœud ne puisse plus assumer ce rôle. Alors, le réseau, ou du moins un sous-ensemble de nœuds, sera définitivement hors d'atteinte. L'*orchestration* engendre une utilisation du réseau de plus en plus concentrée sur un nombre inversement restreint de liens de communication. Cette organisation centralisé provoque une latence non négligeable causée par la longueur plus importante des chemins utilisés [99].

De son coté, la *chorégraphie*, en délégant les décisions sur des élément répartis du réseau, diffuse les flux de communication de façon plus homogène sur l'ensemble de la structure. Elle se fonde sur la dissémination des échanges et permet une consommation plus régulière de la bande passante. La structure complète devient moins dépendante des pannes, dont les impacts peuvent être circonscrits à une zone géographique. Les liens de communication les plus sensibles (ceux les plus proches du nœud central, dont la perte altère la structure même du réseau) sont moins systématiquement mobilisés par l'application. Cependant, d'autres problèmes apparaissent : l'application à réaliser ne se prête pas forcement à une forme répartie de coopération de services [78] [41]. En l'absence de point central, le contrôle du déroulement de l'application, ou encore les tests de celle-ci deviennent plus complexes à accomplir, notamment lorsque des décisions doivent être prises suite à des événements extérieurs au nœud analysé. Enfin, puisque chaque partie prenante n'a pas de vue d'ensemble du processus global, les échanges perdus peuvent entraîner des réactions

en chaîne, et l'application aboutir à un état instable ou incohérent, tout en gardant une apparence de bon fonctionnement.

3.2.4 L'approche services et WSAN

Parce qu'il offre des capacités de traitement et un accès réseau, chacun des objets mis en jeu dans notre cadre applicatif procure plus que des données. Les informations détenues par les objets sont susceptibles d'être mises à disposition grâce à des services qui peuvent devenir de plus en plus évolués. Cette approche *Services pour les objets* a été notamment proposée par Z. Shelby [117], et est de plus en plus communément admise (D.B. Green [67], A. Broring [37], A. Resguy [111] et C. Buckl [39]). S. Duquennoy a construit des serveurs web sur capteurs [57] et A. Pintus organise des orchestrations de services web dans les WSAN [103]. Une grande similitude dans l'approche de l'organisation des échanges existe entre les WSAN et la SOA. En effet, l'opposition entre les modes automatisé et semi-automatisé des WSAN renvoie à celles des approches chorégraphiées et orchestrées de la SOA. Dans le premier cas, la *chorégraphie* reproduit les échanges directs entre nœuds sans centralisation, alors que l'*orchestration*, avec son point de contrôle unique et dominant, peut être associé au rôle prépondérant du puits (et de l'unité de traitement logique qui se trouve plus loin derrière) du mode *semi-automatisé*. C'est pourquoi, dans la suite de notre exposé, nous considérerons l'un et l'autre comme très proches et interchangeables. Construire une *chorégraphie* de services implémentés sur des capteurs permet d'obtenir le mode *automatisé*, tandis que l'*orchestration* équivaut au mode *semi-automatisé* des WSAN.

Cette équivalence, au cœur de notre étude, nous amène maintenant à nous intéresser à la vérification de l'efficacité du mode automatisé (donc des chorégraphies) telle qu'elle est présentée par Y. Akyldiz dans [19]. Si les chorégraphies sont plus pertinentes, quel est réellement leur apport sur les WSAN ?

3.3 Étude théorique comparative des deux approches

Pour quantifier l'impact des architectures logicielles sur la structure réseau WSAN et évaluer les éventuels gains, nous avons procédé en deux étapes. Tout d'abord, nous avons construit une formulation mathématique du problème. Les résultats obtenus à partir d'une étude statistique nous ont permis de valider l'analyse mathématique. Ensuite, des expériences sur notre plateforme sont venues compléter l'étude théorique en témoignant des gains obtenus sur une application réelle.

3.3.1 Approche du problème

Nous nous sommes principalement intéressés à la circulation des messages dans le réseau WSAN. Nous savons que les messages sont émis par les capteurs, tandis que les effecteurs sont les destinataires d'ordres résultant le plus souvent de l'analyse des mesures des capteurs. D'autre part, dans les solutions les plus répandues (ZigBee [12]

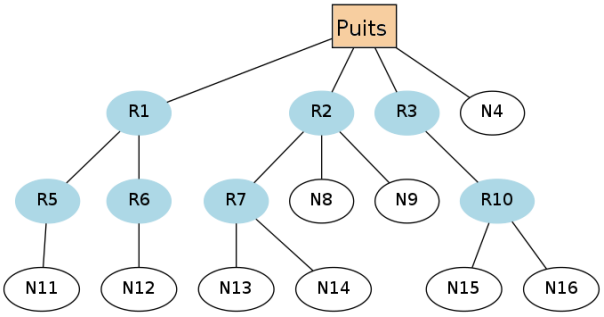

FIGURE 3.5 Un réseau représenté sous la forme d'un arbre. Dans cet exemple, la longueur du chemin entre le nœud N9 et le nœud N13 est de :
1) pour une *chorégraphie* : 3 sauts (R2-R7-N13)
2) pour une *orchestration* : 5 sauts (R2-*puits*-R2-R7-N13)

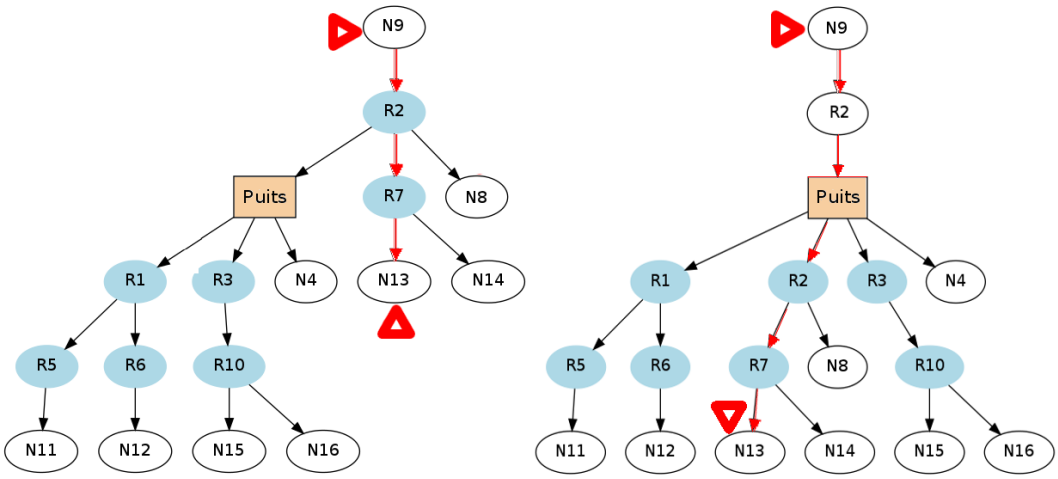

FIGURE 3.6 De son point de vue, le nœud semble être la racine d'un arbre différent. Sur cette figure est représenté le chemin du nœud 9 au nœud 13 tel que vu par le nœud 9 dans le cadre d'une *chorégraphie* (à gauche) ou celui d'une *orchestration* (à droite)

et 6LowPAN [85]), la communication entre les nœuds est organisée sous la forme d'un arbre. Dans les deux cas, le *puits* est la racine de l'arbre.

Pour notre étude, nous allons donc étudier la circulation des messages entre les différents nœuds d'un arbre selon deux modes de circulation :

en passant toujours par la racine de l'arbre, pour matérialiser l'architecture *orchestrée* ou "*semi-automatisée*" ;

en utilisant le chemin le plus direct entre les nœuds afin de reproduire l'architecture *chorégraphiée* ou "*automatisée*".

La détermination de la longueur du chemin utilisé servira à comparer le coût de l'acheminement dans chacune des deux organisations, et à estimer le gain offert par la meilleure solution. La longueur des chemins est calculée pour l'ensemble des couples de nœuds possibles du réseau, et ce afin de ne privilégier aucun cas particulier, tout en simulant une communication très générique. En effet, le rôle de chaque élément

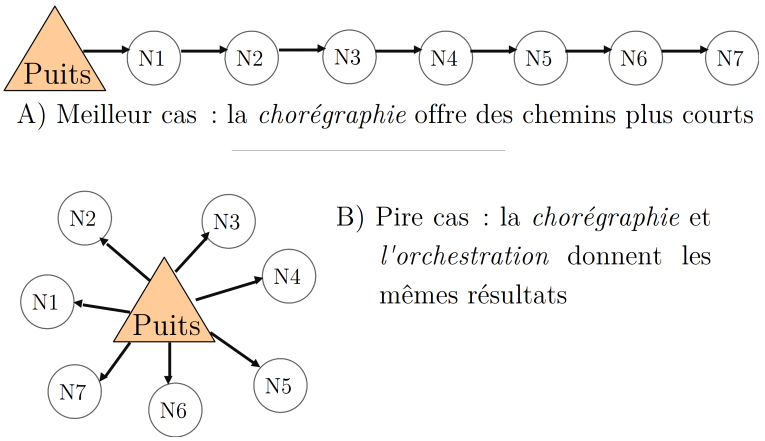

Figure 3.7 Les deux formes d'arbres caractéristiques donnant les meilleur et pire cas. Un arbre dans lequel chaque profondeur ne contient qu'un nœud offre un gain optimal pour une utilisation selon l'approche *chorégraphiée* (cas A). A l'inverse, la *chorégraphie* sur un arbre d'une profondeur de 1 (tous les nœuds en lien direct avec la racine) n'obtient aucun gain par rapport à l'*orchestration* (cas B).

est indépendant de sa position dans l'arbre, sauf pour le puits(*sink*) qui en est la racine. Chaque nœud, en tant que feuille de l'arbre, relaye les messages vers ses destinataires s'il y a lieu (principe de l'organisation de l'acheminement en multi-saut). La longueur du chemin entre deux nœuds est évaluée en nombre de sauts (figure 3.5). De ce point de vue, et selon nos deux approches logicielles, la structure complète peut être considérée comme un arbre spécifique à partir du nœud émetteur (un nœud voit lui aussi l'ensemble du réseau comme un arbre dont il est la racine). La figure 3.6 présente cette vision au regard de la *chorégraphie* et de l'*orchestration*. Ces arbres serviront pour l'évaluation de la longueur de l'ensemble des différents chemins.

3.3.2 Cas extrêmes : le meilleur et le pire cas

Nous avons déterminé les cas extrêmes produisant des résultats significatifs lors de la comparaison des effets de l'architecture applicative sur un arbre (voir figure 3.7). Dans cette figure, les nœuds de l'arbre sont numérotés de 0 (qui représente le *puits*) à n (dernier nœud). La longueur du chemin entre deux nœuds i et j (avec $0 < i < n$ et $0 < j < n$) s'évalue en nombre de sauts.

Étudions tout d'abord le cas du "*modèle en ligne*" de la figure 3.7-A, dans lequel le nœud 7 n'est accessible qu'en passant successivement par les nœud 6, 5, 4 etc. L'addition de la longueur de tous les chemins possibles (c'est-à-dire pour tous les i et les j possibles avec $i \neq j$) divisée par le nombre total de couples (i, j) donne la longueur moyenne des chemins entre les n nœuds (le nombre total de couples est divisé par deux pour éviter de compter le couple (i, j) et (j, i), car notre formule de somme ne fait apparaître que les couples uniques). Nous obtenons alors la formule

générique suivante :

$$\mu_(n) = \frac{2}{n(n-1)} \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} distance_{i,j}$$

Cette formule générique se simplifie :

$$\mu_(n) = \frac{2}{|P|} \sum_{P} distance_{i,j}$$

avec $P =$ ensemble des couples possibles de n nœuds.

Il s'agit maintenant de calculer la distance entre i et j. Dans le cas de l'*orchestration*, le chemin part du nœud i, va jusqu'au *puits* (nœud 0), puis revient dans le sens inverse jusqu'au nœud j. La distance entre les deux nœuds *orchestrés* i et j correspond donc à la somme de leurs indices $distance_{i,j} = (i + j)$. Le message fait i sauts pour revenir au *puits*, puis j sauts vers l'effecteur. Pour l'orchestration, on obtient donc une distance moyenne de :

$$\mu_o(n) = \frac{2}{|P|} \sum_{P} (i + j) \tag{3.1}$$

En ce qui concerne la *chorégraphie*, le calcul de la distance s'évalue en cherchant le plus court chemin (parmi ceux de notre arbre) entre les deux nœuds i et j. La valeur obtenue prend la forme $distance_{i,j} = (j - i)$, car il s'agit d'un arbre "*en ligne*". Ce qui nous donne le résultat suivant dans la formule générique :

$$\mu_c(n) = \frac{2}{|P|} \sum_{P} (j - i) \tag{3.2}$$

Lorsqu'elles sont simplifiées, les formules (3.1) et (3.2) révèlent que, dans le cas extrême d'un arbre en ligne de n nœuds, la longueur moyenne des chemins utilisés par l'*orchestration* et la *chorégraphie* sont respectivement :

$$\mu_o(n) = n + 1$$

$$\mu_c(n) = \frac{1}{3}(n + 1)$$

Dans un arbre "en ligne", la longueur moyenne parcourue par un message d'une application *chorégraphiée* est d'un tiers de celle parcourue lorsque l'application est *orchestrée*.

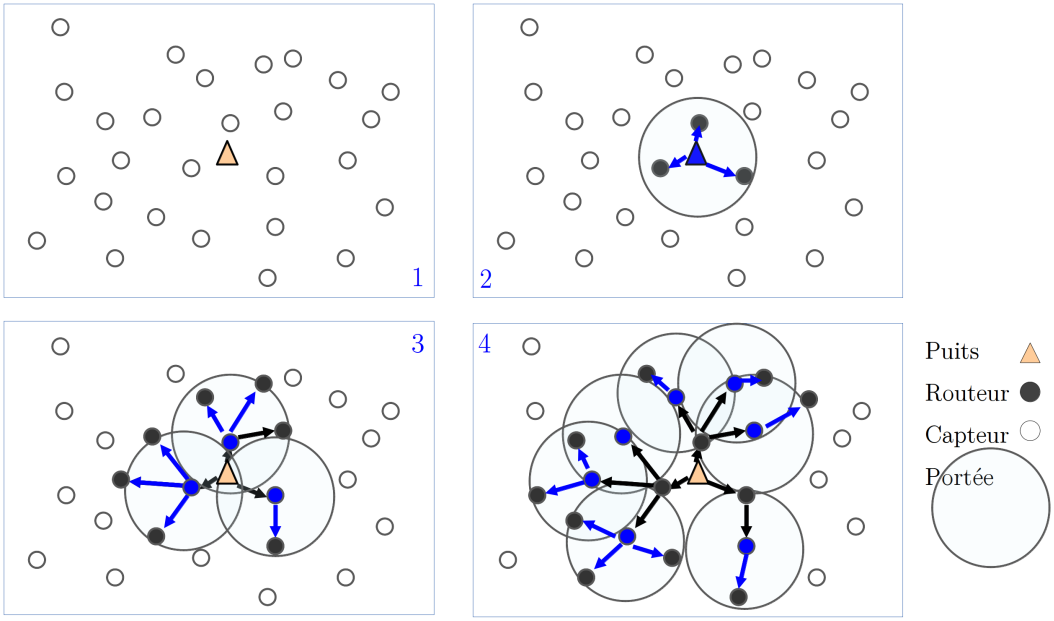

Figure 3.8 Notre algorithme pour construire l'arbre se déroule de la façon suivante : le *puits* (au centre) recherche ses voisins à portée (cercle gris) et les enregistre. À leur tour, ceux-ci cherchent les voisins à portée, et ainsi de suite. Chaque nœud mémorise ses fils, et l'arbre se construit par propagation.

Par contre, dans le cas d'un arbre de profondeur 1, dans lequel tous les nœuds sont à 1 saut du *puits* qui joue le rôle de racine, (voir figure 3.7-B), la longueur des chemins pour n'importe lequel des couples de nœuds (i, j) est toujours de 2. L'utilisation d'un arbre de ce type donne une parfaite égalité au regard de l'organisation logicielle, la *chorégraphie* n'apportant alors aucune amélioration par rapport à l'*orchestration*.

Ces deux cas sont évidemment extrêmes et ne se rencontrent jamais dans la réalité. Ils fournissent cependant les bornes entre lesquelles se situe l'amélioration escomptée sur la longueur moyenne des chemins entre deux nœuds quelconques par la *chorégraphie* par rapport à l'*orchestration* : celle-ci varie de "gain nul" (cas d'un arbre de profondeur 1) à 2/3 (offert par la *chorégraphie* par rapport à l'*orchestration* sur un arbre "*en ligne*"). Pour toutes les autres formes d'arbre, la *chorégraphie* produit une longueur moyenne de chemin plus courte que l'*orchestration*, comprise entre ces deux valeurs extrêmes.

3.3.3 Étude statistique des cas intermédiaires

La réalité de l'organisation des réseaux de capteurs impliqués dans l'Internet des objets se situe quelque part entre ces deux cas extrêmes (meilleur et pire cas de la figure 3.7). Notre étude statistique évalue les gains offerts par la *chorégraphie* par rapport à l'*orchestration* sur des configurations d'arbres aléatoires.

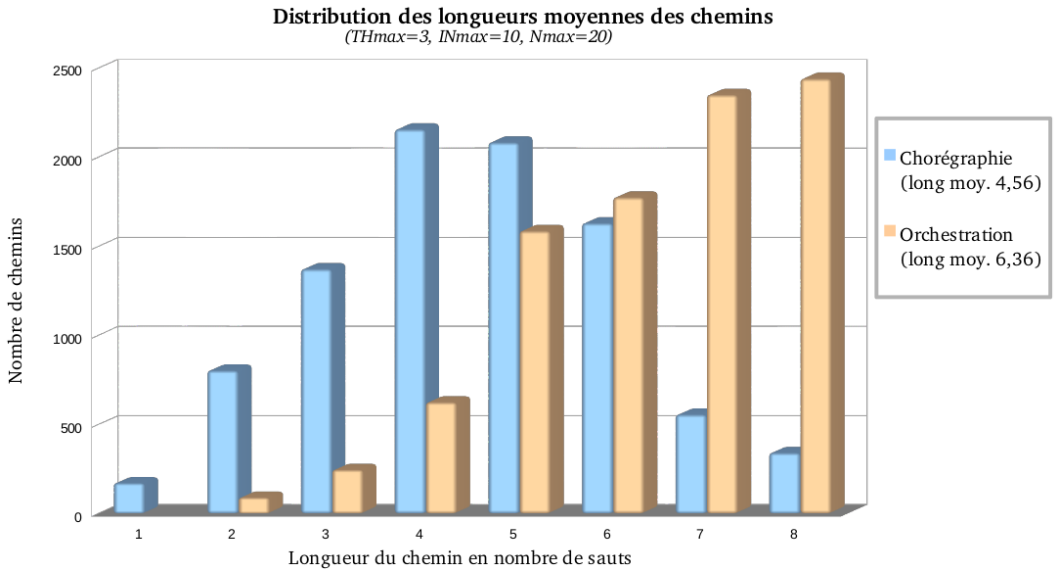

FIGURE *3.9* Comparaison de la longueur des chemins selon le mode *chorégraphié* et le mode *orchestré* sur un arbre très large et de hauteur limitée (profondeur 3).

Pour cette étude statistique, nous avons aléatoirement positionné des nœuds dans un carré (voir figure 3.8). Le *puits* (S) se situe arbitrairement au centre de ce carré. La portée radio est définie grâce à un rayon donné en paramètre. L'accessibilité des nœuds entre eux est déterminée selon l'UDG (*Unit Disk Graph*, Graphe de Disque-unité). Un nœud est considéré comme "à portée" si sa distance est inférieure au rayon passé en paramètre, et inaccessible dans le cas contraire. Les valeurs choisies pour les expériences l'ont été empiriquement afin d'obtenir régulièrement des arbres suffisamment différents, et dans lequel l'ensemble des nœuds sont connectés. En l'occurrence, l'espace de positionnement est un carré de 10∗10, et le rayon UDG vaut 3. Le *puits* au centre commence par rechercher les nœuds accessibles. Ces nœuds à leur tour retrouvent les nœuds atteignables (voir figure 3.8). Les effets de l'utilisation de la *chorégraphie* et de l'*orchestration* sur l'arbre résultant sont analysés selon la longueur des chemins entre tous les couples de nœuds. Notre simulation produit la distribution du nombre total de chemins, classés par longueur, dans chacun des deux cas utilisés pour le parcours dans l'arbre (*orchestration* et *chorégraphie*).

Les figures 3.9, 3.10 et 3.11 présentent les résultats de nos comparaisons entre les 2 modes sur des arbres construits selon les 3 paramètres suivants :

la hauteur maximale de l'arbre (**THmax**) ;
le nombre maximum de nœuds internes (**INmax**) ;
le nombre de nœuds maximum (**Nmax**).

La hauteur permet de limiter le nombre de sauts. Le nombre maximum de nœuds indique le nombre de fils que pourra accepter un nœud. Le nombre de nœuds internes correspond au nombre de fils ayant le droit d'avoir eux aussi des fils (c'est-à-dire le nombre de nœuds n'étant pas des feuilles).

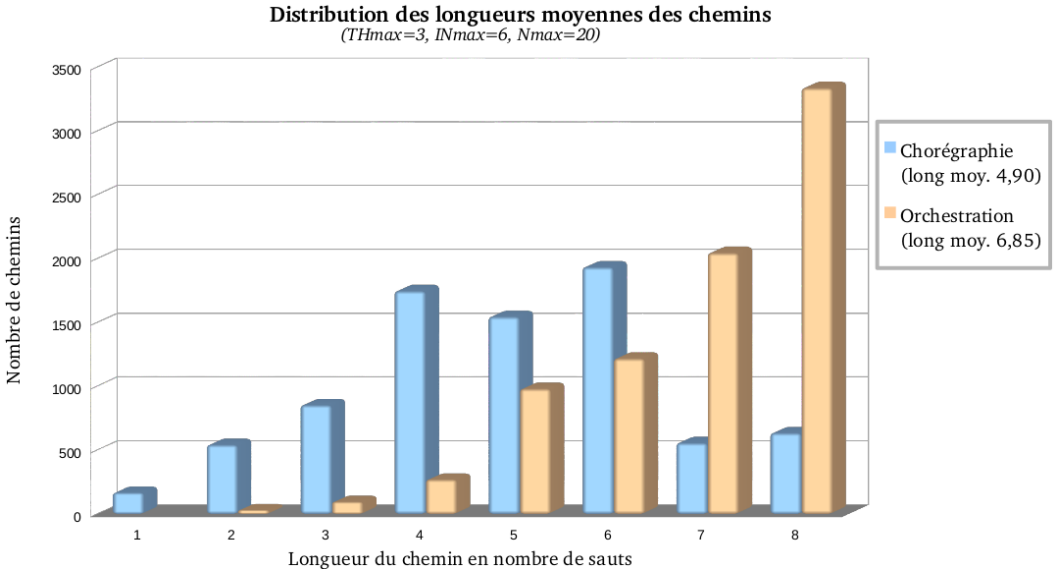

Figure *3.10* Comparaison de la longueur des chemins selon le mode *chorégraphié* et le mode *orchestré* sur un arbre dont la construction est pilotée par les valeurs données dans les exemples ZigBee.

Nous avons fait varier ces paramètres afin de construire des arbres de différentes formes. Ces paramètres, qui influent sur la structure des arbres, sont directement à l'œuvre dans certains réseaux WSAN. ZigBee, par exemple, exploite des variables équivalentes (*Cmax*, *Lmax* et *Rmax*) pour construire sa topologie. Dans ZigBee, le *puits* est le Coordinateur (la racine), le *Lmax* correspond à notre **THmax**, le *Rmax* est le nombre de routeurs (nombre de fils qui peuvent avoir des fils, notre **INmax**) et enfin *Cmax* le nombre total de fils (notre **Nmax**). Dans 6LowPAN, l'algorithme RPL [127] (*IPv6 Routing Protocol for Low power and Lossy Networks*) définit lui-même la structure de son arbre. Notre étude se voulant indépendante de toute implémentation, ces paramètres sont nommés de façon générique.

Toutes nos simulations sont basées sur un ensemble de 100 nœuds. Pour chacune des simulations, nous avons fait varier les trois paramètres de construction de l'arbre afin d'obtenir un panorama complet. Pour chacun des arbres obtenus, la longueur de tous les chemins en partant de chaque nœud selon les deux organisations, *chorégraphie* et *orchestration* a été calculée. Nous avons établi la distribution moyenne des longueurs de chemins pour 1000 arbres construits selon les paramètres indiqués. Chaque expérience a été reproduite 10 fois, et les résultats moyennés avant d'être reportés sur les figures 3.9, 3.10 et 3.11.

Notre première figure 3.9 donne la distribution de la longueur des chemins entre deux nœuds pour un arbre d'une hauteur réduite et assez large (hauteur limitée à 3, 20 fils maximum par nœud dont 10 routeurs au plus). Même si ce type d'arbres n'avantage pas a priori particulièrement les *chorégraphies*, les chemins obtenus sont généralement et significativement plus courts pour ce mode. Lorsqu'on ramène ces résultats à la réalité des réseaux de capteurs, il en résultera un meilleur temps de

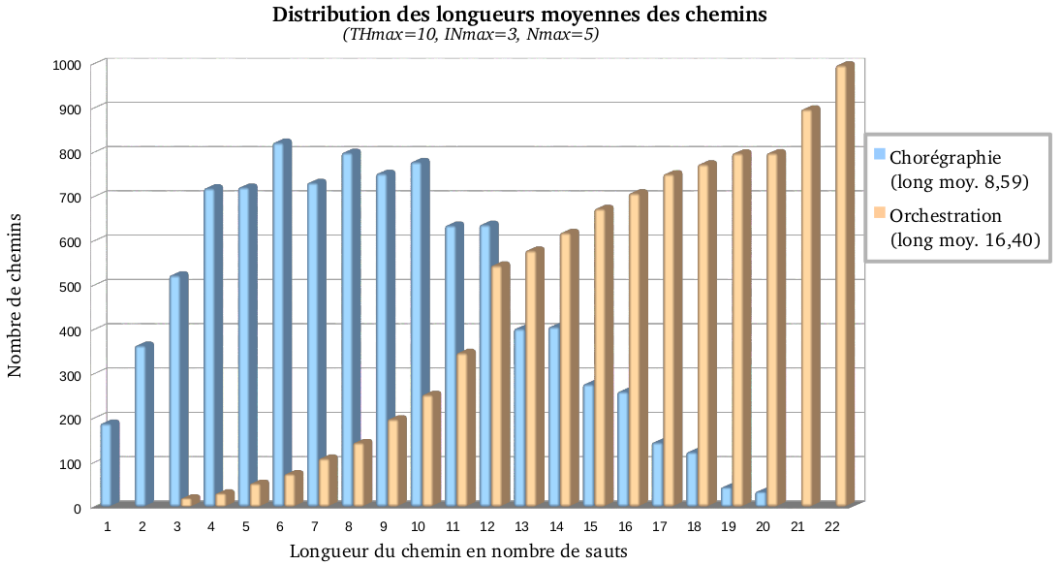

Figure *3.11* Comparaison des deux modes d'organisation applicative sur un arbre haut et peu large, favorable à la *chorégraphie*.

réponse, une réduction de la consommation globale d'énergie dans le réseau et un moindre risque de pertes d'informations liées à la fiabilité relative des transmissions.

Notre deuxième figure 3.10 montre la distribution des longueurs de chemins pour un arbre construit selon les paramètres utilisés par défaut dans les démonstrations de ZigBee ($THmax$=3, $INmax$=6 et $Nmax$=20). L'arbre résultant est moins compacté que le précédent, car chaque nœud accepte un nombre inférieur de routeurs parmi ses fils. Nos évaluations font apparaître une augmentation significative du nombre de chemins atteignant la longueur maximum dans le cas de l'*orchestration*. Puisque le nombre de routeurs diminue à chaque niveau, les nœuds étant massivement repoussés dans la périphérie, au pourtour de l'arbre. L'*orchestration* devient plus pénalisante en terme de charge pour le réseau.

La dernière figure 3.11 révèle les atouts de la *chorégraphie* lorsque la topologie du réseau lui devient favorable. Une hauteur importante de l'arbre, peu de routeurs et de nœuds à chaque niveau génèrent une structure très ramifiée qui offre une grande variété de longueurs de chemins, ce qui est propice à l'usage de la *chorégraphie*.

L'ensemble des simulations menées nous a fourni les différentes moyennes reportées dans la table 3.1. Nous avons calculé l'écart-type afin de vérifier la pertinence d'une comparaison des moyennes. Sa valeur est très faible (inférieure à 1% dans la grande majorité des cas) et ce, malgré nos 1000 tirages. L'étude donne des résultats très stables. Si on observe la table 3.1, les valeurs obtenues par l'étude statistique sont cohérentes avec celles issues de l'analyse mathématique. Les résultats de la table 3.1 sont classés selon la forme de l'arbre. Les valeurs sont bien comprises entre les deux bornes des cas extrêmes présentés dans le paragraphe précédent.

Ces différentes expériences montrent que pour tout type d'arbres, une exploitation selon une logique *chorégraphiée* permet d'utiliser, pour les mêmes besoins de

Table *3.1* Moyenne théorique de la longueur des chemins au regard de l'approche de l'architecture logicielle

Réseau de n nœuds	Longueur moyenne du chemin		
	Orch.	Chor.	Orch./Chor.
THmax=0 INmax=0 Nmax=n (*Pire cas*)	2	2	1.00
THmax=3 INmax=3 Nmax=10	6.96	5.67	1.23
THmax=3 INmax=10 Nmax=20	6.36	4.56	1.39
THmax=3 INmax=6 Nmax=20	6.85	4.90	1.40
THmax=10 INmax=3 Nmax=5	16.40	8.59	1.92
THmax=n INmax=1 Nmax=1 (*Meilleur cas*)	(n+1)	1/3(n+1)	3.00

communication, un ensemble de chemins dont la longueur moyenne est significativement plus courte que ceux proposés par l'*orchestration*. Cette amélioration varie d'un rapport de 1 à 3, selon la forme de l'arbre.

3.4 Approche expérimentale et résultats

Les analyses mathématique et statistique ci-dessus quantifient les apports de la *chorégraphie* sur une structure arborescente. Puisque la transmission est la fonction la plus consommatrice d'énergie [139] [23] [96] alors que l'énergie est la contrainte majeure des réseaux de capteurs, tout gain en termes de longueur moyenne des chemins utilisés aura un impact significatif et prolongera la durée de vie du WSAN. Dans cette partie, nous allons expérimenter les effets des deux organisations sur un réseau réel afin de vérifier la validité des résultats obtenus lors de l'étude théorique des effets de l'approche logicielle.

3.4.1 Description de l'expérience

Cette expérience sur banc de test cherche à mesurer l'impact des architectures logicielles sur la topologie d'un réseau de capteurs. Dans ce but, nous avons développé un logiciel spécifique tournant sous Contiki [55]. Contiki est un système d'exploitation libre pour différents types de matériels utilisés pour la recherche, tels que les TelosB, MicaZ [2] ou encore Sensinode [3]. Contiki est distribué avec l'émulateur Cooja [4] pour tester la validité des codes produits. L'émulation matériel proposée par Cooja nous semble préférable à la simulation du comportement sous NS2. L'objectif est d'obtenir des résultats les plus proches possible de la réalité, et le code de test est utilisable sur du matériel réel. L'émulateur Cooja nous permet cependant de multiplier les expériences, de disposer d'autant d'instances que nécessaire, et de facilement récupérer des informations sur le comportement des nœuds.

2. Fournis par Memsic http://www.memsic.com/.
3. Site web de Sensinode http://www.sensinode.com/.
4. Cooja émule le matériel et simule le réseau entre les différentes instances d'objets émulés.

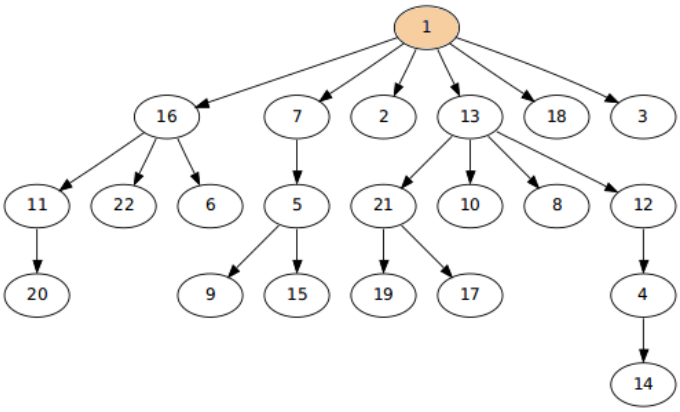

Figure *3.12* Topologie du réseau auto-générée par RPL lors de l'expérience n°50 concernant l'*orchestration*. Les routeurs de plus haut niveau sont le 16, 7 et 13.

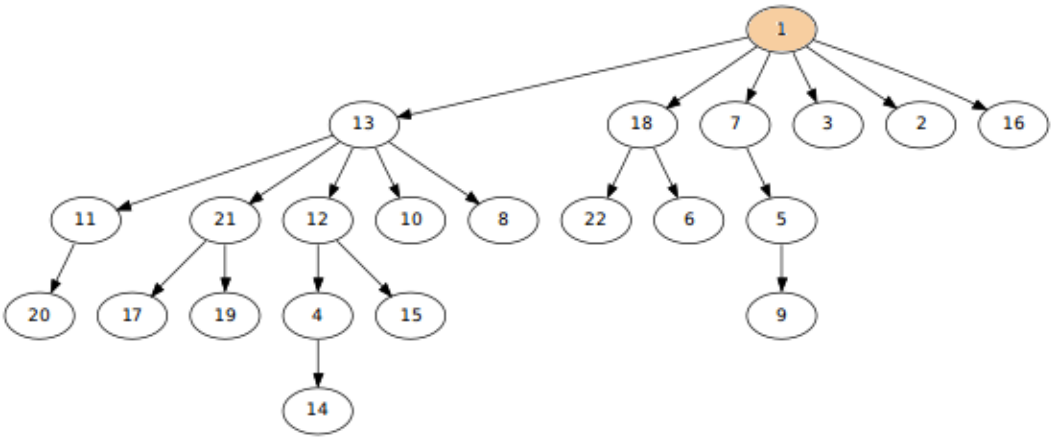

Figure *3.13* Topologie auto-générée lors de l'expérience n°60 : le nœud 16 n'est plus un routeur, le nœud 18 est chargé d'acheminer les messages pour les nœuds 22 et 6. Enfin, le rôle du nœud 13 est devenu plus important.

Contiki met en œuvre une implémentation de 6LowPAN (IPv6 natif pour les objets [118]) et son protocole de routage RPL [127]. Dans le protocole RPL, actuellement à l'état de brouillon IETF, la transmission des données est organisée grâce à une topologie en arbre, chaque nœud retransmettant vers son père ou un de ses fils selon les besoins. Contrairement à ZigBee, l'arbre RPL est auto-organisé par les nœuds eux-mêmes et peut se reconfigurer dynamiquement au cours de l'utilisation, sans possibilité de le contrôler, de le figer, en toute autonomie. Le fonctionnement de RPL lors de notre 50^{e} expérience, tracé dans la figure 3.12, avait évolué quelques heures plus tard, lors de notre 60^{e} expérience (figure 3.13). L'arbre peut varier à tout moment, même à l'intérieur d'une expérience. Cette versatilité n'est pas gênante puisqu'au contraire, nous cherchons à obtenir une image réelle des impacts des organisations logicielles sur les réseaux.

Lors de nos expérimentations, nous avons extrait plusieurs métriques : outre les fils de chaque nœud (ce qui nous a permis de reconstruire les arbres donnés figures 3.12 et 3.13), nous avons récupéré le nombre de paquets IP envoyés, reçus, et relayés, représentant l'activité globale du réseau (que ces paquets aient été générés pour notre application de mesure ou pour l'organisation propre au réseau). D'un autre coté, notre programme de tests recense le nombre de messages qu'il a effectivement émis et reçus, donnant une vue purement applicative de l'activité. Aussi, la comparaison des deux ensembles de valeurs nous informe sur les impacts de l'organisation logicielle par rapport à l'activité globale du réseau. Pour cette expérience, nous utiliserons le terme *paquet* pour ce qui concerne l'activité générale du réseau (incluant donc le coût de son organisation), et plutôt le terme *message* lorsque nous nous référerons à notre programme de test *chorégraphie / orchestration* (c'est-à-dire au niveau applicatif).

Durant l'étude théorique, nous avions systématiquement calculé la longueur de tous les chemins possibles, en prenant tous les nœuds 2 à 2. L'ensemble des couples nous avait donné un résultat exhaustif. Mais ici, dans le cadre d'une vérification de l'impact sur un environnement réel, nous avons préféré limiter le nombre d'interlocuteurs pour un nœud. Contrairement à ce que nous avons fait dans l'étude théorique, le choix d'un unique interlocuteur nous a semblé plus réaliste. En effet, il semble peu probable que, dans une application Internet des objets, l'ensemble des nœuds interagissent avec *tous* les autres. Notre vision des applications IoT nous amène à considérer les interactions entre petits îlots plus plausibles. Mais puisqu'il est difficile de construire un modèle représentatif d'une telle construction, nous avons multiplié les expériences en créant des dépendances aléatoires entre des nœuds. En moyennant tous les résultats obtenus, nous disposons d'une vision générale crédible.

Le déroulement de chaque test a suivi la démarche suivante :

1. Chaque nœud choisit un unique destinataire aléatoire, ceci afin de représenter le lien entre un capteur et un effecteur. À noter qu'un même nœud peut être choisi par plusieurs autres émetteurs, et que les rôles ne sont pas figés. Un nœud peut être cible de plusieurs autres, et lui-même dialoguer avec un effecteur. Ces cas sont en phase avec notre vision de l'Internet des objets.
2. Une fois l'interlocuteur choisi, le nœud attend un temps aléatoire (entre 0 et 59 secondes) puis lui envoie 30 messages, à intervalle régulier (1 par minute). Le message transmis fait une taille d'une trentaine de caractères.
3. Pendant ce temps, il compte les messages reçus (ce nœud a pu être choisi comme cible par 0 à $n-1$ nœuds).
4. Retour au point 2 de l'algorithme.

Notre évaluation consiste à comparer la vue *applicative* (nombre de messages envoyés et reçus par chaque nœud) et la vue *réseau* (nombre de paquets réellement reçus, émis et relayés). Les expériences existent sous 2 variantes :

tous les messages doivent passer par la racine de l'arbre : c'est l'*orchestration/mode semi-automatisé* ;

les messages vont directement au destinataire : c'est la *chorégraphie/mode automatisé*.

L'objectif est de mesurer le gain apporté par la *chorégraphie* sur une topologie en arbre complètement aléatoire, et qui ne lui est pas spécifiquement favorable (arbre non contrôlé et couples de nœuds choisis au hasard).

Trois programmes ont été développés pour Contiki 2.5.

Pour chaque nœud :

un client/serveur *chorégraphié* : alors qu'il écoute les autres, ce nœud émet vers le destinataire choisi ;

un client serveur *orchestré* : il n'émet que vers le *puits* et n'écoute que celui-ci. L'indicatif du destinataire se trouve dans le message transmis.

Sur le *puits* :

Chorégraphie : c'est le programme fourni dans Contiki 2.5 (*examples/ipv6*), plus précisément celui de l'exemple rpl-udp (démonstration RPL gérant le nœud racine et passerelle vers l'extérieur). Aucune modification n'a été apportée à ce programme, qui se charge notamment d'initialiser la construction de l'arbre RPL (c'est pourquoi il n'est pas compté dans la liste des nœuds de notre application).

Orchestration : il s'agit d'une version du programme ci-dessus à laquelle nous avons ajouté une analyse du contenu du message reçu, une extraction de l'indicatif du destinataire, puis la retransmission du contenu au nœud indiqué. Ce nœud joue alors exactement le rôle du centre de décision de l'*orchestration*. En théorie, ce "chef d'orchestre" est situé plus loin dans le réseau global, mais les débits des autres réseaux sont suffisamment supérieurs à ceux des WSANs pour que l'impact de leur traversée soit négligeable dans notre expérience. Un vrai centre de décision distant pénaliserait encore un peu plus l'*orchestration*.

Nous avons utilisé 21 nœuds plus un *puits* pour chacune des expérimentations. Pour obtenir des résultats significatifs et comparables, les nœuds ont été positionnés aléatoirement dans l'espace lors de l'étape de préparation, puis figés pour l'ensemble des tests. Les expériences ont été lancées 100 fois pour chacune des deux architectures logicielles. La seule différence entre les deux groupes d'expériences réside dans la version du programme qui établit la communication entre les deux nœuds en passant par le nœud central ou bien en direct.

3.4.2 Quantification des gains de la *chorégraphie*

Nous avons fait de nombreux tests et éliminé les résultats non significatifs. L'organisation de notre expérimentation (création aléatoire de liens entre deux nœuds) exige de régulièrement modifier les couples de nœuds s'échangeant des messages afin de lisser les résultats. Les expériences répétées 100 fois qui ne font varier ni l'algorithme, ni la position des nœuds, mais uniquement le destinataire introduisent la diversité nécessaire. L'utilisation d'un aléa pour le choix du partenaire reflète que, dans la réalité, rien ne prouve qu'un capteur et un effecteur soient effectivement

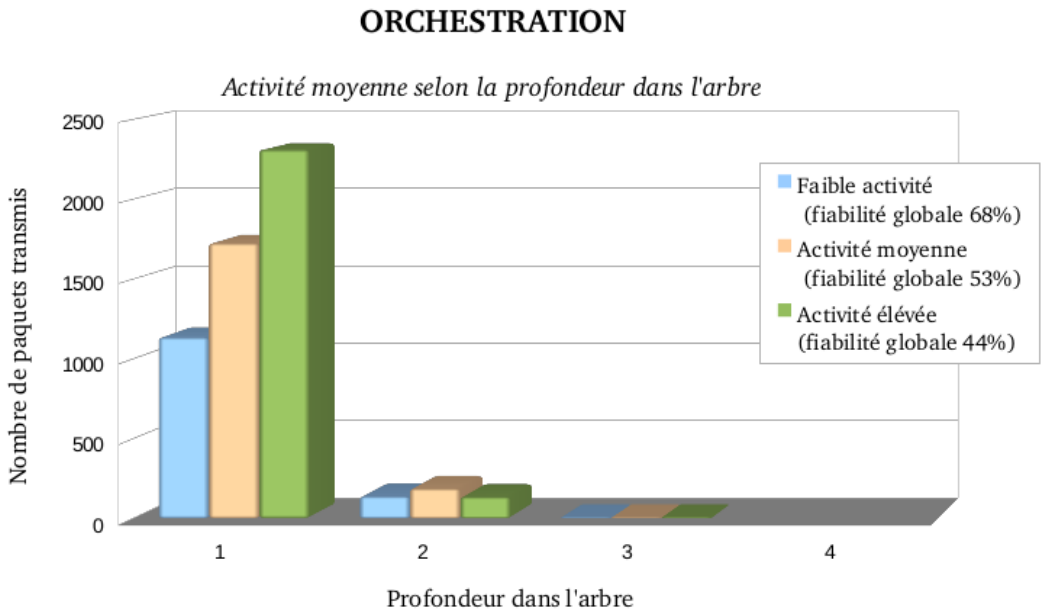

Figure 3.14 L'*orchestration* entraîne un usage intensif des routeurs situés tout en haute de l'arbre, puisque tous les messages remontent au point central, puis redescendent vers un nœud.

proches l'un de l'autre, même si cela est plausible. Et même s'ils le sont géographiquement, le parcours à suivre dans le réseau peut, lui, être bien différent (nœuds 15 et 19 dans la figure 3.12 par exemple). D'autre part, un arbre RPL est amené à évoluer dans le temps sans qu'on puisse le contrôler. Dans ce cadre, de multiples tirages aléatoires d'un interlocuteur procurent des conditions expérimentales appropriées et représentatives.

Nous avons aussi éliminé les premiers résultats dans notre analyse, car la construction de l'arbre RPL aurait pu fausser les résultats. Ce surcoût organisationnel, lié à l'organisation des couches plus basses est surtout sensible lors du démarrage du réseau, et affecte d'égale manière les deux organisations logicielles. Cette surcharge est ponctuelle, et concentrée sur les premiers échanges. Elle n'est pas représentative de l'activité normale, une fois l'arbre construit. Nous avons gardé les deux derniers tiers de l'ensemble des mesures afin d'en extraire les tendances générales une fois l'environnement stabilisé.

Afin d'en avoir une vue plus fine, les résultats ont été regroupés dans trois classes correspondant à la charge constatée sur les routeurs proches du *puits* (par exemple les nœuds 13, 18 et 7 sur figure 3.13). Ces trois classes réunissent les résultats pour une activité de relais *faible*, *moyenne*, et *élevée* des nœuds de niveau 1. L'activité est estimée en nombre total de paquets relayés par l'ensemble des nœuds à 1 saut du *puits*. La totalité des expériences retenues est agrégée dans ces trois classes. Pour chacune de ces classes, nous collectons l'*activité réseau* constatée à chaque niveau de profondeur dans l'arbre, ainsi qu'une mesure de la *fiabilité globale du réseau*, c'est-à-dire le pourcentage de l'ensemble des messages reçus par rapport au nombre de

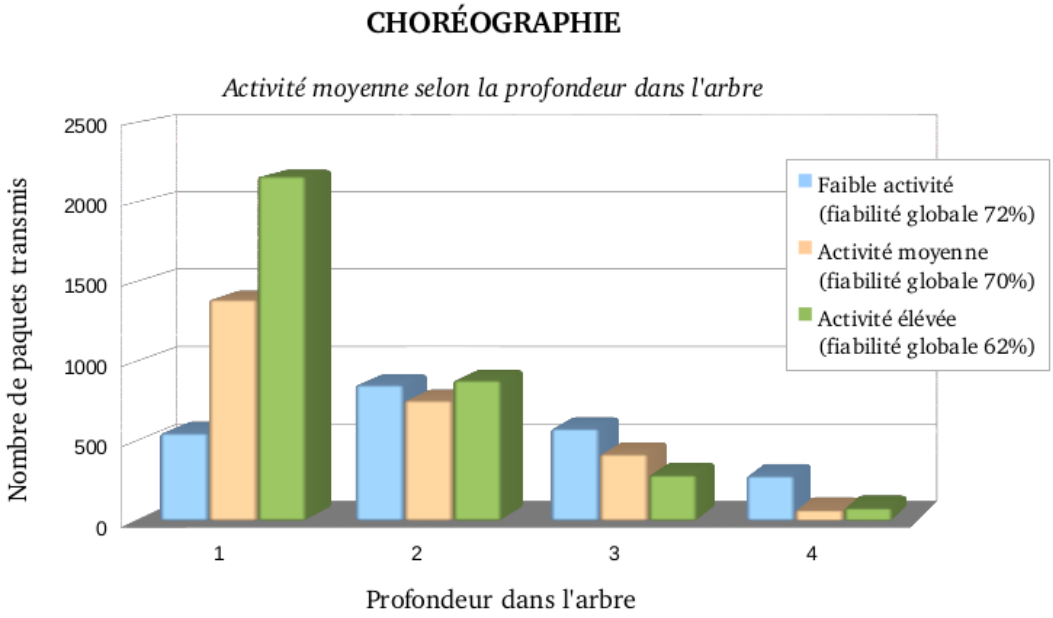

FIGURE *3.15* Dans une chorégraphie, le trafic est mieux réparti sur l'ensemble des profondeurs de l'arbre. La diversité des chemins empruntés par les messages permet d'atteindre un niveau de fiabilité supérieur, car cette architecture favorise moins la formation de goulets d'étranglement.

messages envoyés. Ainsi nous obtenons une vue globale de la charge du réseau, au regard du niveau d'activité des nœuds à 1 saut, selon les deux modes d'organisation (*orchestrée* ou *chorégraphiée*).

L'impact de l'*orchestration* sur l'activité au sein du réseau est illustré sur la figure 3.14. Chacune des barres indique le nombre de paquets en sortie des nœuds, regroupés selon leur profondeur dans l'arbre. Le niveau d'activité est visualisé selon la position du nœud. La majorité du trafic se concentre sur les nœuds les plus proches du puits. Dès qu'on s'en éloigne, l'activité chute de façon spectaculaire. Elle est quasi nulle à la profondeur 3. À ce constat s'ajoute la piètre fiabilité du réseau. Les taux de pertes sont importants, au mieux 68% de messages arrivent à destination, et au pire plus de la moitié des messages sont perdus. Les messages n'étant pas relayés par le niveau 1, saturé, l'activité des niveaux 2 et 3 reste faible. Dans le cas de l'*orchestration*, plus un nœud est haut dans l'arbre, plus le trafic qu'il doit gérer est important. Comme annoncé dans [99] [106] [19], l'*orchestration* occasionne une plus grande charge sur le réseau, et notamment en concentrant un usage important sur les relais de haut niveau.

La chorégraphie quant à elle (figure 3.15) produit un profil très différent. Pour le même niveau d'activité des nœuds de rang 1, un plus grand nombre de paquets peut être échangé aux niveaux inférieurs. La fiabilité du réseau s'accroît. Pour une activité toujours aussi élevée en niveau 1, les pertes de messages sont moins nombreuses, car certains chemins évitent les nœuds centraux. Les nœuds situés à des profondeurs intermédiaires sont plus sollicités car les trajets empruntés ne convergent plus systé-

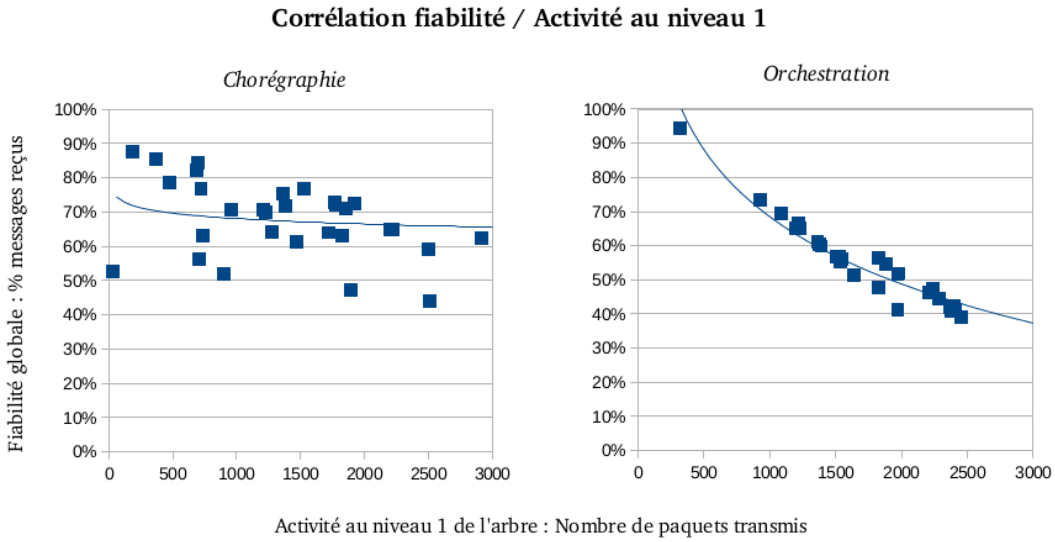

Figure *3.16* Recherche d'une corrélation entre la charge des routeurs de niveau 1 et la fiabilité générale de l'*orchestration*. Plus le niveau 1 est chargé, moins le réseau est fiable. Des messages sont perdus à tous les niveaux. L'*orchestration* sature les nœuds proches du *puits*. S'agissant de la *chorégraphie*, la recherche de corrélation ne donne pas de résultats probants. Le réseau fonctionne en général mieux, et la fiabilité varie indépendamment de l'activité. Elle est plutôt liée aux destinations des messages qu'à leur nombre global dans le réseau.

matiquement vers les nœuds de haut niveau. Globalement, un nombre plus important de messages circule. Les échanges sont diversifiés, et la charge de transmission mieux répartie sur la topologie.

L'étude de l'ensemble des données collectées lors de nos expériences révèle des différences importantes sur le pourcentage de messages arrivant à destination selon les 2 modes. Nous avons recherché s'il existait des liens entre la charge des nœuds voisins du *puits* et la fiabilité générale constatée sur le réseau. Afin de vérifier cette hypothèse selon laquelle la *chorégraphie* exploite mieux l'infrastructure du réseau, les fiabilités globales constatées sont exprimées au regard de l'activité des nœuds de niveau 1 (figure 3.16). Notre expérience est très consommatrice en ressources car les nœuds émettent de l'information très régulièrement et génèrent un fort trafic. Le réseau est volontairement saturé, afin d'approcher une situation génératrice de pertes, et obtenir des résultats discriminants.

Dans le cas de l'*orchestration*, le lien entre le niveau d'activité des nœuds de profondeur 1 et la fiabilité globale du réseau est net, et le dépassement des capacités offertes par le réseau (ou plus exactement la surcharge des éléments les plus exposés) conduit à une dégradation de l'ensemble de l'application. La figure 3.16 trace la tendance globale. Les petites variations sont dues à la gestion interne, notamment les vérifications et reconstructions imprévisibles de l'arbre (voir figure 3.13 et 3.12), liées au coût organisationnel de la topologie. Pour la *chorégraphie* au contraire, l'étude ne révèle pas de lien entre l'activité et la fiabilité. Les nœuds centraux ne

sont pas impliqués dans la totalité des échanges, et le trafic est mieux réparti. La fiabilité globale du réseau atteint de meilleurs niveaux, et l'application est moins sensible à l'activité relevée au niveau 1.

3.5 Conclusion

L'extension de l'Internet à l'univers des réseaux de capteurs nous a poussés à nous interroger sur l'impact des technologies applicatives sur les contraintes très spécifiques des WSAN. En cherchant à savoir si ceux-ci étaient sensibles à l'organisation logicielle choisie, nous nous sommes intéressés à la comparaison des effets d'une *orchestration* et d'une *chorégraphie* de services sur tout type d'infrastructures en forme d'arbre, sachant que les protocoles réseaux WSAN utilisés dans le cadre de cette thèse utilisaient cette forme de réseau.

Plusieurs études [19] [136] indiquent qu'intuitivement le choix de la *chorégraphie*, quand il est possible, devrait être le meilleur en terme de longueur de chemins. Nous avons proposé ici une étude mathématique pour quantifier le gain espéré et repérer les cas extrêmes en donnant le meilleur et le pire cas. En l'occurrence, nous avons déterminé que la longueur moyenne des chemins d'un arbre peut être jusqu'à 3 fois plus courte lorsqu'on utilise un mode *chorégraphié* par rapport au mode *orchestré*. Ces meilleur et pire cas dépendent de la forme de l'arbre : en ligne pour le meilleur, avec un profondeur de 1 pour le pire.

Nous avons ensuite mené une étude statistique sur des arbres variés, en utilisant un mode de construction de l'arbre utilisant le même dispositif que celui utilisé dans ZigBee [12]. Les résultats sont fidèles à ceux de notre quantification, compris entre les deux bornes, et variant selon la forme de l'arbre en conformité avec nos cas extrêmes.

Enfin, une expérience sur banc de test nous a permis de constater que l'effet des deux organisations logicielles est significatif pour une même application. Même si d'autres messages, non liés à l'application circulent sur le réseau (par exemple, des messages de contrôle), le réseau est plus fiable, lors de la montée en charge, quand l'application est organisée en mode *chorégraphiée* plutôt qu'*orchestrée*.

Si elle est préférable sur un réseau fortement contraint en énergie comme les WSAN, la *chorégraphie* génère alors des problèmes à un autre niveau. Elle demande une implication plus importante de chacun des nœuds, la possibilité de disposer de codes exécutables localisés, et de prises de décision au plus proche. Pourtant, ces matériels peuvent être très divers (que ce soit en termes de puissance, d'autonomie, de simplicité de mise à jour, mais aussi de système d'exploitation, de langage de programmation et de protocoles réseau) et le tout doit s'organiser harmonieusement. L'hétérogénéité des parties prenantes qui constituent l'Internet des objets, dans leur incapacité à proposer une plateforme commune pour le développement, pénalise la création d'applications ayant recours à la chorégraphie, pourtant préférable pour certains des réseaux mis en œuvre en son sein.

Notre prochain chapitre s'intéresse à une solution d'unification de la vision des différents éléments pour fournir un environnement commun de développement. Une plateforme homogène, installée sur l'ensemble des nœuds, présente au programmeur une uniformité propice à la portabilité de ses codes, à leur réutilisation et aux échanges. A la fois simple à programmer et léger, notre framework D-LITe donne à chaque objet impliqué dans l'IoT une apparence générique et universelle.

Chapitre 4

D-LITe : l'objet virtualisé reprogrammable

Sommaire

> Un homme d'expérience ne devrait jamais s'égarer dans le concret. L'abstrait reste l'âme des affaires.
>
> Quand passent les faisans
> Michel Audiard

De 50 à 100 Milliards d'objets connectés à l'Internet [124] et un flux de communication M2M (*Machine-To-Machine*) 30 fois supérieur à celui d'êtres humain à machine en 2020, l'Internet des objets promet de multiples modifications de ce que nous connaissons aujourd'hui du *réseau des réseaux*. Dans le futur, l'Internet des PCs ne sera plus majoritaire sur l'ensemble des échanges réalisés au regard du trafic généré par les objets entre eux [121].

Nombre d'études de l'Internet des objets [117][103][67][95][111][39][24][84] préconisent les architectures orientées Services (SOA). Les WSAN sont l'un des constituants majeurs de l'IoT [90], et leur mode *automatisé/semi-automatisé* font écho aux organisations *Chorégraphiée/Orchestrée* des architectures SOA. Le chapitre précédent a montré que l'utilisation du mode *automatisé* présente des atouts non négligeables en terme de respect des contraintes énergétiques des WSAN, et qu'une approche *Chorégraphiée* d'applications Internet des objets est pertinente. Cependant, ces prévisions de l'essor de l'IoT se heurtent, aujourd'hui, à l'hétérogénéité des matériels mis en œuvre. Les tentatives de création d'un architecture de services SOA sur les réseaux de capteurs doit faire face à une "*tour de Babel*" constituée de matériels, langages de programmation, API et systèmes d'exploitation incompatibles entre eux. Quelle que soit l'approche empruntée, cette "*tour de Babel*" génère un frein important au développement d'outils et d'applications, car une telle diversité de composants cantonne chaque réalisation à un secteur limité du marché, avec peu voire aucune possibilités d'adaptation et d'ouverture aux solutions alternatives des autres acteurs. Cette absence d'interopérabilité engendre un manque de visibilité préjudiciable au développement du domaine [124] [31] [39].

Afin d'assurer l'interopérabilité, *middleware* et *machines virtuelles* proposent des solutions d'interconnexion et d'échanges entre éléments très différents. Un *middleware* (ou *intergiciel* en français) est un mot valise décrivant une couche logicielle intercalée entre, d'un coté, une application et de l'autre, l'élément à gérer (figure 4.1). Il peut s'agir d'un matériel, d'une autre application voire d'une base de données [50] [33]. L'intergiciel, médiateur chargé de l'adaptation et de la réalisation de la communication [66] [25] [84] [109] [75] et [124] implique un effort non neutre de chacun des partenaires : l'installation du middleware sur l'élément qui recevra les demandes, et l'emploi de commandes spécifiques pour l'appel aux services distants dans la partie utilisatrice (figure 4.1). De son coté, la *machine virtuelle* est une solution logicielle qui s'installe sur l'hôte. Mais à la différence des intergiciels qui imposent l'appel explicite à leurs fonctionnalités, la machine virtuelle exécute en tant que tel le programme d'origine, qui n'a aucune connaissance de l'artifice. Il ne s'agit pas d'une collection de fonctions qui peuvent être appelées/manipulés à distance (API), mais bel et bien de l'émulation d'un matériel sur laquelle l'application

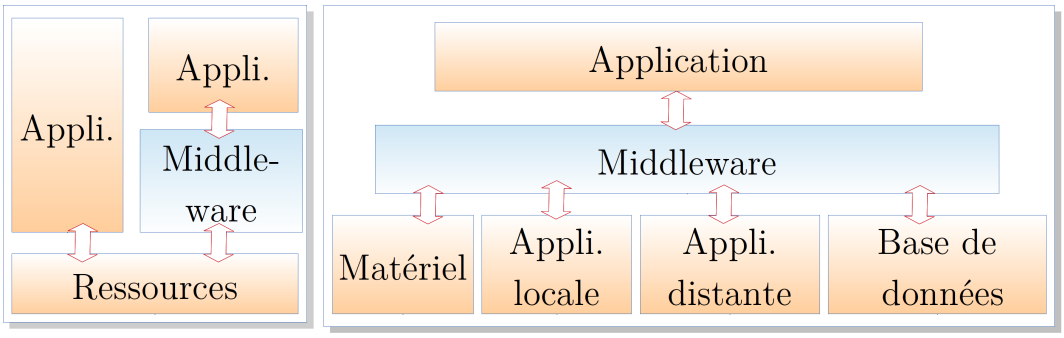

Figure *4.1* Un intergiciel (ou *middleware*) permet l'adaptation et l'accès à des ressources en offrant un accessibilité simplifiée. Utile pour découpler une application des outils qu'elle utilise, l'intergiciel permet l'accès à des applications distantes, à des outils de persistance ou à du matériel.

s'exécute, en utilisant les ressources locales à travers l'illusion unifiée offerte par la virtualisation [123]. Aussi l'application sera-t-elle développée uniquement à destination de la machine simulée peu importe la machine réelle. Cependant, proposer une telle universalité engendre souvent une couche logicielle d'une taille imposante, et c'est la raison pour laquelle les solutions standards telle la machine virtuelle Java ne peuvent prétendre résoudre la problèmatique (si ce n'est dans des versions très restreintes [38]).

Afin d'homogénéiser et faciliter les interactions entre les différents éléments tout en masquant leurs spécificités matérielles, nous avons défini et réalisé une machine virtuelle spécifiquement conçue pour les besoins logiciels et matériels de l'IoT, adaptée à ce type d'applications, d'une taille réduite, générique et optimisée par rapport aux contraintes rencontrées. Ce chapitre présente **D-LITe** (*Distributed Logic for Internet of Things sErvices*), notre framework pour objets communicants. D-LITe est une machine virtuelle qui utilise un langage spécifique appelé **SALT** pour *Simple Application description using Transducers for Internet of Things*. SALT permet à un programmeur d'exprimer la logique de chaque élément de son application sous forme de machine à états finis tout en permettant d'exploiter l'abstraction offerte par la machine virtuelle D-LITe. Pilotable à distance dans le respect de l'approche REST, D-LITe est adapté à l'IoT car sa programmation, sous forme de transducteurs [1] à l'expressivité étendue, génére des codes conformes à ses contraintes, concits et facilement téléchargeables sur les objets. Les caractéristiques de cet ensemble répond aux besoins de la programmation d'applications distribuées multi-plateformes dans le cadre de l'Internet des objets. Mais la fiabilité de certains réseaux à l'œuvre dans l'IoT reste limitée (les WSAN notamment), et des erreurs peuvent apparaître et entraîner des incohérences. Ce chapitre se clôt sur notre solution intégrée à SALT permettant la remise en cohérence des applications. L'efficacité des resynchronisations est évaluée par rapport à leur surcoût en terme de trafic.

1. Automate à états finis agrémenté d'un alphabet en sortie.

4.1 Programmation des objets : SOA, REST, FST et abstraction matérielle

Permettre à de multiples éléments autonomes d'interagir grâce à leurs capacités de traitement et de communication plaide pour une organisation distribuée de la logique des applications à réaliser. Une démarche naturelle consiste à appréhender chaque élément, chacun des objets, comme offrant des services [135], à charge ensuite d'organiser la communication, la combinaison et les interactions entre les différents services disponibles [69].

4.1.1 Applicabilité de l'architecture orienté services dans l'IoT

L'Architecture Orientée Services (SOA, *Services Oriented Architecture*) décrit une approche organisationnelle des méthodes de programmation dans laquelle une application utilise les services rendus par d'autres entités logiques, le plus souvent distantes et à visibilité contrôlée, dans un mode distribué [59] [101]. L'application résultante est composée du traitement des résultats des briques applicatives mises en œuvre. L'organisation globale est assez flexible, et les services peuvent évoluer indépendamment les uns des autres, sans mettre en péril l'ensemble. De multiples services peuvent être fournis, avec ou sans autorisation, utilisables par de multiples clients différents ou réservés à un seul, avec des visibilités différenciées selon l'habilitation des consommateurs. Ce type d'organisations trouve son intérêt dans l'interpénétration possible des systèmes d'informations, de façon très souple, en limitant les contraintes, ce que traduit le terme "*couplage faible*" [129].

La SOA est construite autour de :

l'*Annuaire* des services ;
la *Description* de chaque service ;
le *Format* des échanges.

L'annuaire des services permet le référencement et la recherche des services proposés par l'ensemble des partenaires. La norme d'annuaire la plus répandue UDDI(*Universal Description Discovery and Integration*) permet le stockage des descriptions d'un service, ainsi que les méta-données utiles au référencement et aux recherches. Ce premier niveau, plutôt sémantique, permet de trouver dans l'annuaire le service correspondant aux besoins. La seconde étape à résoudre concerne un aspect plus fonctionnel, c'est la description du service qui permet au programme utilisateur de vérifier les modalités de l'appel : *Nom* du service, *description*, *adresse*, nombre et type des *arguments*, nombre et type des *valeurs de retour*. La standardisation d'un langage de description (DL) permet de comprendre le format d'appel du service et comment y accéder (WSDL [7] pour les Web Services par exemple). Le format des échanges doit lui aussi respecter certaines contraintes afin de garantir la fiabilité et le contrôle des contenus. SOAP [9] (ancien acronyme de *Simple Object Access Protocol*) définit le contenu et la présentation de ces messages (voir figure 4.2). Les échanges s'effectuent souvent grâce au protocole HTTP (mais d'autres sont éventuellement proposés, tel SMTP, ou en utilisant des sockets réseau), et le contenu obéit

Figure 4.2 Un message SOAP est une enveloppe qui contient une entête et un corps. Dans cet exemple adapté de la RFC de SOAP [9], l'entête indique un message de type *alertcontrol*, et le corps du message contient le message "*Élise sort de l'école*".

aux règles du langage balisé XML. L'ensemble constitue une solution puissante et paramétrable.

L'approche service dispose d'atouts [67], mais les contraintes de l'IoT (une chaîne est limitée aux caractéristiques du maillon le moins puissant) ne plaident pas pour un protocole aussi verbeux que SOAP [68]. La versatilité des cas pris en compte entraîne de multiples définitions et descriptions qui en alourdissent le contenu. Ce reproche, assez communément exprimé [62] et [142], prend d'autant plus d'importance ici que les réseaux de capteurs, part non négligeable de l'Internet des objets, proposent une charge utile très réduite, et sont soumis à d'importantes contraintes énergétiques dissuadant l'usage de protocoles aussi loquaces [37].

À y regarder de plus près, le système d'annuaire UDDI est sur-dimensionné par rapport aux besoins de l'IoT. Dans ce cadre, la liste des objets susceptibles d'être intégrés dans une application (domotique, ville intelligente, etc.) est souvent limitée, et le ratio entre objets impliqués/disponibles bien supérieur à ce qu'on trouve dans la SOA classique de l'Internet des PC (dans lequel une application, si complexe soit-elle, n'utilise qu'une infime part de l'ensemble des services disponibles). Si l'idée de faire des recherches du service approprié dans la multitude offertes par la SOA sur l'Internet fait sens (l'usage d'un annuaire général y est pertinent) [53], la spécificité des applications IoT favorise plutôt l'obtention directe et exhaustive des objets directement accessibles et intégrables (liste limitée, dont une grande partie est connue à l'avance, puisqu'il s'agit des objets propres à l'utilisateur).

Un service d'annuaire aussi complet UDDI se justifie moins pour l'IoT sauf à vouloir y intégrer les services offerts par la SOA classique. Quant à leur description des services via WSDL, la charge de traitement de cette description exprimée en XML devient pénalisante au regard des capacités limitées des objets, et leur incapacité à embarquer un analyser XML capable d'en extraire les informations utiles. Une solution IoT doit proposer des informations équivalentes à celle de la SOA clas-

sique, mais en tenant compte des limites importantes des capacités de traitements des nœuds et des contraintes particulières du domaine.

4.1.2 Services REST

SOAP (*Service Oriented Architecture Protocol*) [9] est un protocole applicatif qui intègre une définition du format des messages permettant à deux applications de s'échanger des requêtes et leurs réponses. Cependant, il doit lui même être transporté dans un protocole applicatif (*HTTP*, voire *SMTP*), ce que la RFC de SOAP définit comme étant de la "*liaison de protocole*" (protocol binding). La construction de l'architecture SOA avec cette solution fait l'objet de critiques : Pourquoi transporter un protocole de couche 7 du modèle OSI dans un protocole de cette même couche (*SOAP* via *HTTP*) ? Ce constat, fait par R. Fielding et R. Taylor, les a conduit à présenter REST [62] : REST (*REpresentational State Transfer*) est un style d'architecture. Puisque HTTP offre déjà un protocole applicatif d'échange de contenu, les auteurs ont recensé les commandes existantes et leur ont fait correspondre les besoins de la SOA. Ainsi, à un service donné est associée une URI pour y accéder. Les ordres HTTP *GET*, *PUT*, *POST* et *DELETE* servent à mimer les actions CRUD (*Create*, *Retrieve*, *Update* et *Delete*). De ce fait, un service est remplacé par une ou plusieurs ressources REST, et l'utilisation de ces ressources s'invoque par les commandes du protocole HTTP. Le protocole HTTP permet, lui aussi, de préciser le type de contenus échangés (grâce au *Content-Type*).

REST est un choix de plus en plus répandu sur l'Internet et se révèle adapté à l'Internet des objets. Sa conformité, son utilisation efficace du protocole HTTP et sa légéreté en font une solution particulièrement indiquée sur les WSAN qui peinent à supporter la verbosité de SOAP [39] (ou seraient pénalisés par une compression des contenus, puisqu'elle impliquerait des calculs à chaque échange) [105]. Des messages aux contenus plus légers sont conformes aux attentes du M2M et aux capacités des objets impliqués.

Standardiser les échanges REST entre objets respectant les contraintes des WSAN est un premier pas. Mais les obstacles qui freinent le développement de l'Internet des objets ne se limitent pas au choix d'une solution SOA. La principale difficulté trouve sa source dans la diversité des matériels impliqués dans les applications. Nous proposons ici de résoudre ce problème grâce aux FST et à l'abstraction matérielle.

4.1.3 Transducteurs à états finis

Les *transducteurs à états finis* (Finite State Transducer, FST) sont une forme particulière de *machines à états finis* [94]. Une machine à états finis (ou *automate*) permet de décrire le comportement d'une machine abstraite face à des stimuli extérieurs et de définir ses réactions. Cet automate dispose d'un certain nombre d'états. La réception de messages le fait passer d'état en état grâce aux transitions qui mènent des uns aux autres. Utilisé notoirement en Traitement Automatique du

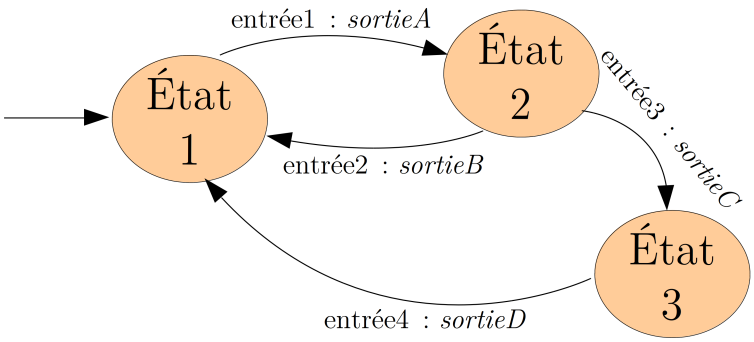

Figure 4.3 Schéma représentant un transducteur : État1 est l'état initial, et les transitions définissent les changements d'états selon les messages entrant. L'entrée *1231434* génère *ABACD*.

Langage (TAL) [79] [80], un transducteur (dont un exemple est donné figure 4.3) est un automate dans lequel le changement d'état, déclenché par un alphabet en entrée, produit un alphabet en sortie. Les transducteurs sont utilisés pour l'application ou la vérification des règles de grammaire dans un texte, ou la reconnaissance de la parole.

Un FST est représenté formellement grâce au 6-uplet $T(Q, \Sigma, \Gamma, I, F, \delta)$. Dans notre cas, les éléments du 6-uplet ont les rôles suivants :

Q représente la liste des *États* du nœud ;
Σ contient l'*Alphabet d'entrée* géré par le nœud ;
Γ contient l'*Alphabet de sortie* que le nœud utilise ;
I contient les *États Initiaux* ;
F contient la liste des *États finaux* (selon notre vision, un FST de l'IoT est sans fin, en continuelle interaction [39]) ;
δ la liste des transitions (nous utiliserons aussi le terme "*liste de règles*").

Les FST obéissent aux mêmes règles et définitions que les automates, à savoir par exemple, être[2] :

complet : tout état propose une transition pour tout élément de l'alphabet ;
déterministe : à partir d'un unique état initial, il existe au plus une transition pour chacun des éléments de l'alphabet, partant de chacun des états ;
asynchrone : l'alphabet en entrée accepte l'élément vide ε, qui permet des transitions automatiques d'état à état.

Les FST utilisés sont déterministes, car notre objectif est d'exprimer un algorithme décrivant la réaction d'un objet à une situation donnée, ce qui interdit tout ambiguïté pour chaque état. Nos FST sont rarement complets puisque chaque état n'a pas obligatoirement à réagir à l'ensemble des stimuli possibles. Enfin, ils sont parfois asynchrones, des actions sont possibles sans qu'un stimulus extérieur n'intervienne (pour préparer par exemple le contexte dans un nœud, au démarrage de l'application).

2. Nous limitons cette liste aux caractéristiques utiles pour cette thèse.

4.1.4 Intérêt des FST pour l'IoT

La raison pour laquelle nous avons choisi les automates tient dans leur capacité à exprimer des algorithmes de façon simple, intuitive et universelle. L'Internet des objets est par essence constitué d'un assemblage de petites unités indépendantes dotées de capacités de traitement réduites [96] [43]. Nous avons vu plus haut que ces objets utilisent de nombreux langages de programmation très techniques, et des système d'exploitation souvent incompatibles entre eux. Ces langages demandent une durée de développement important pour la réalisation de l'application voulue [44]. Les solutions de plus haut niveau couvrent une bonne partie des besoins, mais ont le défaut de ne pas être universelles. Les ZigBee profiles [13], par exemple, se déclinent selon les usages. Mais comment les faire interagir avec des nœuds 6LowPAN et des services du cloud par exemple ? Et si des passerelles existent, jusqu'à quel point implémentent-elles la totalité des fonctionnalités requises ?

L'Internet des objets se fonde sur l'intégration d'un ensemble hétéroclite. L'universalité de la solution proposée pour faciliter cette intégration est primordiale. Notre intérêt pour les FST trouve sa source dans la relative facilité à implémenter l'interpréteur dans chaque type d'objets. Bien que limitée dans son expressivité, l'utilisation des FST n'est pas pénalisante ici car la logique des actions/réactions des objets entre eux semble de prime abord plutôt triviale à décrire. La solution d'offrir un framework pour fédérer la diversité des composants de l'Internet des objets a déjà été promue [22]. Mais comparé aux autres frameworks ([22] [63]), un interpréteur de FST peut être plus facilement supporté par les différents objets (légèreté du code et de son exécution), tout en ne pénalisant pas trop l'utilisateur (décrire ses besoins sous forme de FST ne demande pas un niveau d'expertise élevée en programmation).

L'usage des FST nous permet de résoudre la problématique suivante : notre approche *chorégraphié* de l'IoT sous-entend une collaboration entre des composants autonomes. Nous pensons qu'il est plus logique et efficace de travailler localement sur les informations connues par cet objet, d'en obtenir une synthèse et seulement ensuite d'émettre une conclusion [18]. Dans l'approche *Services chorégraphiés* que nous proposons, un résultat calculé, agrégé et synthétique offre plus d'intérêts que les informations qui ont permis cette déduction [124]. Savoir qu'il faut allumer le chauffage est plus riche de sens qu'une température envoyée toutes les minutes, et une alerte "*parking plein*" permet une réaction bien plus directe que le flux irrégulier du nombre brut de places utilisées.

Grâce à l'utilisation des FST pour exprimer la logique de chacun des objet impliqués, l'algorithme local de traitement des données s'exprime plus simplement et reste concis, universel et multiplateforme. La capacité des FST à réagir aux messages entrants pour en générer en sortie construit une chorégraphie entre les différents objets, chacun réagissant à son environnement, recevant des messages d'autres éléments sollicités par l'application, et en générant pour ceux à l'écoute [96].

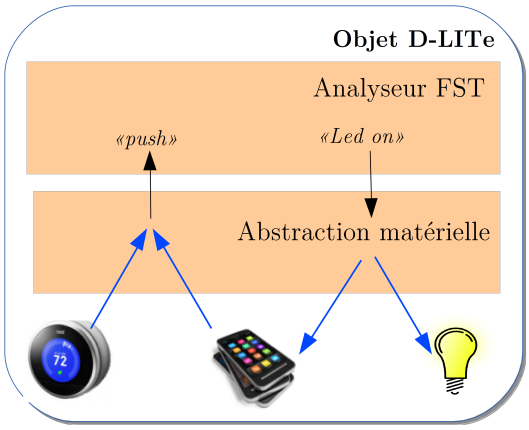

Figure 4.4 Grâce à la couche d'abstraction matérielle, les FST utilisent des mots génériques, et ceux-ci sont transformés et adaptés aux commandes réelles de l'objet. "*led on*" allume aussi bien une lampe que la led d'un téléphone. L'appui d'une touche du smartphone ou d'un interrupteur génère un message générique "*push*" à l'entrée du FST.

4.1.5 Abstraction matérielle

Offrir un langage universel n'est pas tout. En effet, l'interaction avec le matériel reste nécessaire, que ce soit pour récupérer des données ou encore déclencher une action. Or, l'accès aux ressources propres d'un matériel implique d'en connaître les fonctions spécifiques. Souvent, cet accès est fourni par un pilote matériel qui simplifie l'appel aux circuits électroniques. Cependant, même simplifiés, ces appels sont propres à chaque produit, à chaque langage de programmation et système d'exploitation. La "*tour de Babel*" refait son apparition, car même des matériels aux usages similaires (par exemple différents capteurs de température) fournissent des fonctions identiques via des commandes incompatibles entre elles.

Afin de résoudre ces appels multiplateformes, notre solution recourt à une couche d'abstraction matérielle (figure 4.4). Cette couche va traduire un message tel que "*led on*" en son appel réel sur la machine. L'ordre "*led on*", généré par le FST, fait partie de son alphabet de sortie. Cette sortie "*led on*" devra être prise en compte par la couche d'abstraction sur le matériel réel. Pour peu que les portages se coordonnent bien, ou qu'ils respectent une grille de référence commune, les commandes comprises par l'interpréteur de FST peuvent être partagées par des matériels similaires. Ainsi, "*led on*" pourra allumer aussi bien un diode fabriquée par un industriel qu'une ampoule d'un fournisseur concurrent. De la même façon, l'appui sur un bouton d'un TelosB [3], d'un Sensinode [4] ou d'un bouton virtuel dessiné sur l'afficheur d'un smartphone générera un message "*push*" à l'entrée de l'interpréteur de FST. Si cela correspond à une transition pour l'état actuel, celle-ci sera déclenchée.

Ce système permet à la fois d'interagir simplement avec le matériel tout en s'abstrayant des différences entre chaque modèle. Le FST décrit les actions et réac-

3. Fournis par Memsic `http://www.memsic.com/`.
4. Site web de Sensinode `http://www.sensinode.com/`.

tions de façon universelle et abstraite, et la couche d'abstraction matérielle adapte les informations attendues ou émises vers le matériel spécifique. L'objet devient interchangeable. Ce sont ses *fonctionnalités génériques* qui sont mises en avant par l'entremise de la couche d'abstraction matérielle.

L'interpréteur de FST et l'introduction de la couche d'abstraction matérielle permet à notre solution d'offrir une certaine universalité pour exprimer les algorithmes simples, correspondants aux usages de l'Internet des objets, tout en offrant un accès facilité à la diversité des fonctionnalités matérielles des éléments effectivement mis en œuvre dans l'application.

Ces notions sont appliquées dans notre solution D-LITE, le framework de développement d'applications chorégraphiées pour l'Internet des objets, que nous présentons dans la section suivante.

4.2 D-LITe, le framework

Dans l'objectif de construire des applications distribuées pour l'Internet des objets, le framework D-LITe (*Distributed Logic for Internet of Things sErvices*) s'intéresse à la communication entre et avec les objets. L'utilisation des ressources de l'objet doit être d'une part simplifiée et standardisée, mais l'accès à cet objet doit aussi être pris en compte. Que ce soit dans la façon de les programmer, de les paramétrer ou d'échanger avec eux, la diversité des parties prenantes offre un panorama de méthodes de configuration trop diversifié. Si certains éléments sont directement accessibles (smartphones, objets personnels du PAN), d'autres nécessitent l'utilisation d'un accès à l'Internet (services web, réseaux sociaux). Quant aux objets de l'environnement, de type capteurs/effecteurs, en l'absence d'une interface utilisateur (clavier, boutons, etc.), ils requièrent souvent l'entremise d'un ordinateur doté d'une connectique et d'un logiciel spécifique pour flasher une mémoire morte, ce qui limite les redéfinitions ou les évolutions du comportement de l'objet. De plus, l'accès physique à l'objet est parfois tout simplement impossible (capteur coulé dans le sol, fixé dans des faux plafonds, etc.).

Ce qui suit présente la façon dont nous avons traité les problèmes d'accès à distance, de re-programmation et d'échanges entre les éléments D-LITe.

4.2.1 D-LITe : la sémantique des messages

Selon notre point de vue, l'Internet des objets, agrégat d'outils très variés, se fédère autour d'une approche *orientée événement*. L'approche *orientée évenement* alliée à une organisation *chorégraphiée* permet à la fois de soulager le réseau tout en offrant une palette plus riche au programmeur pour la construction de sa logique applicative. Nous avions évoqué plus haut la teneur sémantique des messages échangés dans D-LITe et l'accès à la couche d'abstraction matérielle, le tout au travers d'alphabets entrant et sortant du FST. Trois types de messages entrants et sortant (figure 4.5) sont supportés :

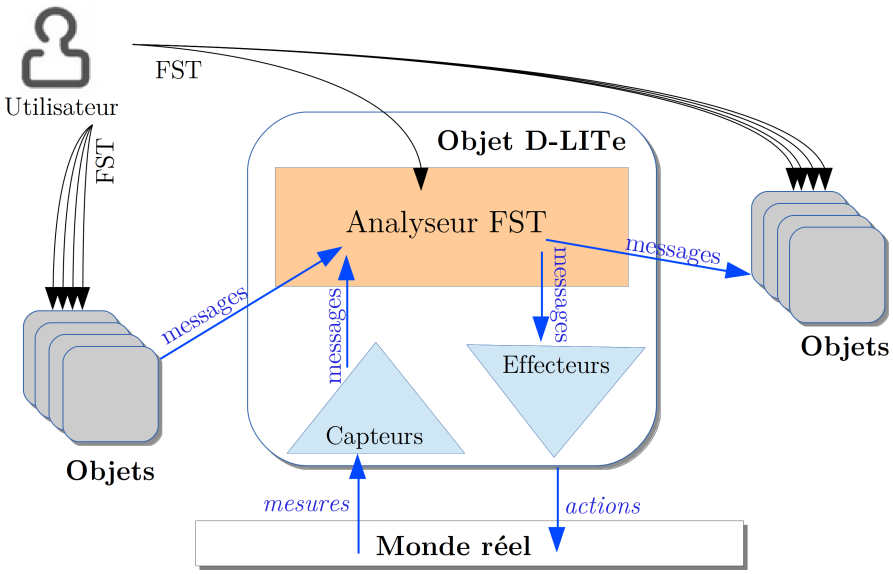

Figure 4.5 Un objet tel que vu par D-LITe. La partie programmable obéit à l'analyseur de FST. Le FST réagit à différents types de messages : Externes (provenant d'autres objets), Hardware ou Logique, (messages internes, provenant ou à destination du matériel).

externe (*external*) : messages échangés entre différents objets, c'est la sortie d'un FST qui devient l'entrée d'un autre (ou plus) exécuté(s) sur une (ou plusieurs autres) machine(s). Ces messages sémantiques informent, sous forme d'événement, les objets abonnés, afin qu'eux-même y réagissent.
matériel (*hardware*) : messages qui proviennent ou sont destinés au matériel. Les messages entrants indiquent un événement détecté par le matériel, qui devient alors un message d'entrée pour l'analyseur FST. Les messages sortants, à destination du matériel, vont déclencher une action.
logique (*logic*) : afin de résoudre des problèmes liés aux limites des FST, nous avons ajouté un module logique et arithmétique à notre machine virtuelle. Ce module peut être piloté via des messages de type logique, afin de créer des variables, y stocker des valeurs et de réaliser des opérations et des tests sur leurs contenus. Ce type de messages peut être vu comme une virtualisation des capacités logiques de l'objet. Ces messages sont considérés comme internes et représentent ses aptitudes à calculer.

4.2.2 Accès distants aux nœuds D-LITe

Puisque notre projet est de permettre la découverte des nœuds, leur reprogrammation à distance et les échanges de messages sémantique entre eux, nous avons besoin de plusieurs accès différenciés à chacun d'eux.

La figure 4.6 décrit l'organisation de D-LITe sur un objet. Sur chaque objet est installé l'interpréteur de FST et la couche d'abstraction pour l'accès au matériel. La seule contrainte de notre proposition concerne sa disponibilité pour un matériel

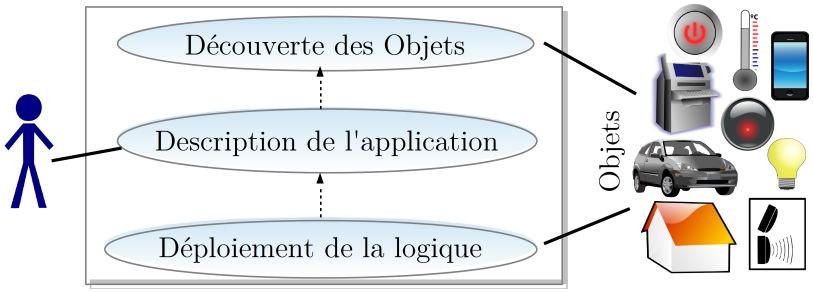

Figure 4.6 Construction d'une application de l'Internet des objets : L'utilisateur découvre ses objets et leurs caractéristiques, décrit l'application dont il a besoin, et celle-ci est déployée afin de s'exécuter.

donné. Actuellement, D-LITe supporte les appareils qui acceptent Contiki-OS (un système d'exploitation libre dédié aux objets de l'IoT), Android (pour tablettes et smartphones), et une API Linux (pour Arduino par exemple, ou tout type de machines exploitables par Linux).

L'interpréteur de FST inclus dans D-LITe est chargé d'obéir à la logique qui lui a été transmise. Cette logique est déposée à volonté par l'utilisateur, à distance, via le réseau. Cet interpréteur écoute et réagit aux messages entrants provenant soit de l'extérieur (par le réseau, et émis par d'autres objets auxquels il est abonné), soit du matériel (de ses capteurs ou de sa partie logique). Ces messages peuvent entraîner une transition du FST, qui déclenche souvent l'émission de messages à destination soit de l'*extérieur* (à la liste de objets qui sont abonnés à celui-ci), ou du *matériel* (action quelconque de l'effecteur, opération logique). Une des options n'exclut pas l'autre, et les messages en sortie peuvent être multiples pour une seule transition.

4.2.3 D-LITe : des services évolutifs

Dans notre approche *orientée services* de l'IoT, chaque objet présent dans une application Internet des objets est accessible en REST. Une des fonctionnalités les plus importantes à nos yeux concerne la possibilité de décrire et modifier à volonté le comportement que chaque objet doit adopter, en lui transmettant la description de sa logique applicative au sein de la chorégraphie.

Les solutions orientées services traditionnelles s'organisent autour des couches applicatives. En SOA, l'accent est mis sur la composition de services figés, offerts par différents prestataires. L'accès au service est la partie visible de celui-ci, selon les modes décrits et définis par son fournisseur. Le service est à consommer en l'état, tel que le fournisseur l'a conçu. Dans l'Internet des PC, le comportement des services est prédéfini et évolue peu dans le temps. Mais l'Internet des objets doit au contraire être bien plus re-configurable. Un même objet doit pouvoir changer de rôle, être utilisé dans une application, puis une autre, parfois se comporter différemment, ou même être appréhendé d'une toute autre manière. L'architecture doit être conçue pour l'évolution ("*design for change*" dans le brouillon de la RFC architecturale de l'IoT [24]).

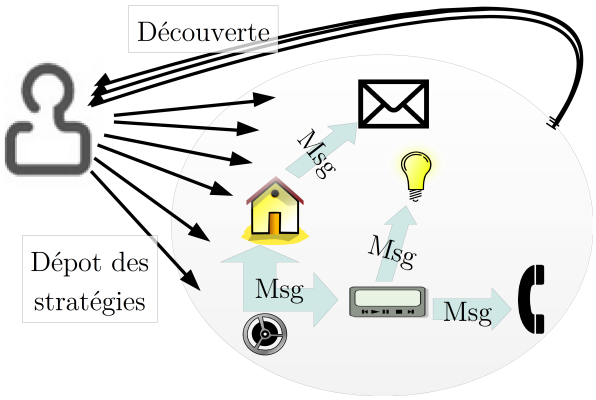

FIGURE 4.7 Utilisation des objets. Un utilisateur découvre les caractéristiques de chacun de ses objets, puis y dépose le service qu'il en attend. Quant aux objet, ils doivent communiquer entre eux.

Cette grande dynamique d'usages exige la proposition de solutions pour le déploiement simple et distant de la logique à exécuter. Contrairement à l'architecture orientée services habituelle, dans laquelle un programmeur construit son application en composant dynamiquement des services rigides et imposés (*bottom-up*, partant de l'existant pour l'agréger), nous proposons une démarche inverse (**top-down**), à la manière de [75] [40] : un utilisateur doit pouvoir construire son application en combinant des services dont il définira le fonctionnement, dynamiquement, à la volée, sur les objets mis en jeu (voir figure 4.6). L'évolution de son application, sa redéfinition, voire sa complète remise en cause pour construire un tout nouvel ensemble se fera toujours de cette façon : *en déposant, à distance, le service attendu*, répondant à ses besoins immédiats, sur les nœuds impliqués dans son application.

4.2.4 Approche top-down

Cette démarche de modification dynamique de la logique embarquée dans un composant s'inspire des concepts offerts par les architectes de la programmation objet, et notamment les Patrons de Conception (Design Pattern, et notamment ceux du *Gang of Four* [65].) Le Patron de Conception utilisé ici est connu sous le nom de "*stratégie*", et permet le changement dynamique du comportement d'un programme, grâce à la délégation de l'exécution du comportement à un élément interchangeable. La transmission d'une nouvelle stratégie entraîne la modification dynamique, en cours d'exécution, du comportement adopté. L'utilisation de "*stratégie*" réalise l'inversion de l'organisation *bottom-up* de la composition de services en une installation *top-down* des services désirés à la volée sur les objets (figure 4.7).

Le téléchargement d'un code à exécuter a déjà été proposé dans les réseaux de capteurs. Des solutions d'*Over-the-Air-Protocol* (OAP) existent [130], mais des problèmes de compatibilité apparaissent, pour les raisons d'hétérogénéité des matériels déjà évoquées. Contrairement aux solutions de OAP (DELUGE [48] et [72], SYNAPSE [112] ou Dynamic TinyOS [98]), notre framework D-LITe se base sur un

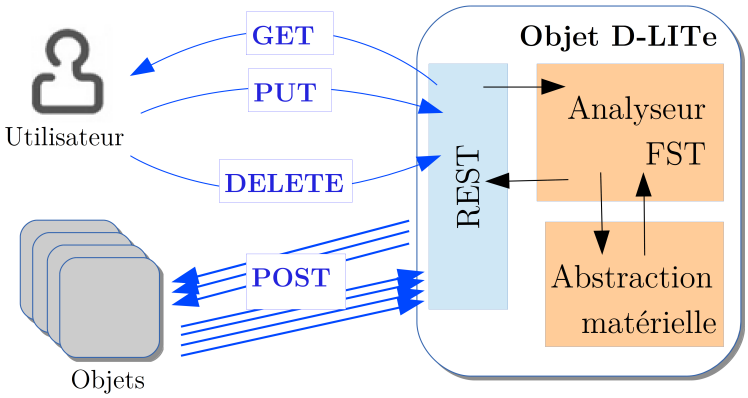

Figure 4.8 Les services REST offerts par un objet. Vu de l'extérieur, celui ci offre la commande GET (découverte), PUT (déploiement de la stratégie) POST(échanges de messages) et DELETE (effacement de la stratégie)

principe plus proche de la solution du Byte-Code. Ici, la *stratégie* qui est transférée est composée du FST et de la liste des objets partenaires, l'ensemble restant concis. Les risques d'erreurs sont minimisés par la taille réduite de l'échange.

Cette organisation donne à notre architecture des aptitudes de déploiement et de redéploiement d'applications, rapides, légères, réactives et adaptées à l'IoT.

4.2.5 Utilisation de REST dans D-LITe

L'Internet des objets prône la ré-utilisation des protocoles robustes et largement adoptés de l'Internet afin d'étendre celui-ci aux objets qui nous entourent. Pour communiquer entre eux, les objets utilisent le protocole réseau *IP* [140] et le protocole applicatif *HTTP* [69]. Suivant les recommandations de R. Fielding sur *REST* [62], *SALT* dispense l'accès à ses services selon 4 méthodes (nom des commandes dans la RFC) de *HTTP*. Les atouts de *REST* pour la construction d'applications *SOA* résident dans sa légèreté et son respect conceptuel du modele *OSI* [142].

Les avantages de *REST* sont multiples. Outre leur correspondance avec nos besoins en terme de services (figure 4.7), sa légéreté par rapport à *SOAP*, *REST* est omniprésent sur l'Internet, garantie d'une interopérabilité importante. Une adaptation aux WSAN, *CoAP* [119] est disponible dans *Contiki-OS*, dans *TinyOS* et dans plusieurs bibliothèques (*libcoap* en C [1], *coapy* en python [2] ou *jcoap* en java [6]) et même en tant que module pour Firefox (*Copper* [3]) pour des échanges interactifs avec les objets.

Aussi utiliserons-nous les commandes *REST* (figure 4.8) de la façon suivante :

GET : découverte des caractéristiques de l'objet, version de la plateforme D-LITe.

PUT : déploiement de la *stratégie*, cette commande utilisée par le programmeur/utilisateur pour déposer son FST sur l'objet. Avant ce PUT, l'objet n'a aucun comportement particulier.

POST : échange de messages entre les objets uniquement. POST transmet la sortie du transducteur à l'entrée de celui des objets abonnés.
DELETE : effacement de la *stratégie*. Lors d'un redéploiement d'application, le programmeur/utilisateur utilise cette commande avant de refaire un PUT.

4.2.6 Implémentations de D-LITe

La première implémentation de D-LITe sur Contiki-OS [55] a été publiée le 17 février 2011. Contiki dispose nativement du support de 6LowPAN [85] et de CoAP [119] pour une interaction directe entre objets et l'Internet. D-LITe a été compilé pour TelosB [5] équipé d'un processeur 16 bits, d'un mémoire morte de 48 Ko et d'une mémoire vive de 10 Ko. Le programme complet, incorporant le support de 6LowPAN et de CoAP, agrémenté de l'interpréteur de FST et des services de découverte, déploiement, communication et effacement (les 4 commandes REST) génère un binaire de 47 Ko qui est flashé sur l'objet. L'objet est alors disponible, pourra se présenter, et offre 4ko de mémoire vive (sur les 10 ko disponibles) pour le stockage de la *stratégie* qui lui sera dynamiquement transmise.

Le code pour Contiki [6] est composé de deux parties principales : d'un coté, un élément commun regroupant l'interpréteur de FST, la gestion de la logique (variables, tests, opérations mathématiques), description du nœud, et de l'autre, une partie spécifique au TelosB chargée du lien entre l'analyseur du FST et les capteurs/effecteurs : l'abstraction matérielle. Ce découpage facilite le portage de D-LITe sur d'autres matériels supportant Contiki-OS. La partie commune reste inchangée, seule l'abstraction matérielle doit être complétée, afin de faire correspondre des messages arbitraires du FST à un appel matériel dans Contiki.

Deux implémentations ont suivi. Tout d'abord une version pour générer des objets virtuels sur Internet (*D-LITe Cloud*), écrite en Java, présente sur nos serveurs, et nous permettant de créer des objets fictifs dotés de leds, de boutons, capables d'afficher des messages... Ensuite, une version Android [7] afin d'inclure smartphones et tablettes dans nos applications, donnant accès à la fois à l'écran tactile de l'appareil, mais aussi à tout une gamme de capteurs (lumière, mouvement) ou d'effecteurs (led dessinée sur l'écran tactile, affichage de message, prise de photo, envoi de SMS, etc...).

Grâce à la généricité de l'abstraction matérielle, le même FST de gestion d'un bouton peut être indifféremment déployé sur tablette, TelosB, smartphone ou objet virtuel. L'approche *services* permet le déploiement et la communication transparente entre les différents éléments. D'autres versions sont en cours de développement, sous forme d'API en C, ou de solution complète en Java, pour l'intégration de nouveaux types d'objets.

Le framework D-LITe offre un accès REST pour la gestion d'un objet générique (découverte de ses capacités, programmation et effacement de son comporte-

5. Fournis par Memsic http://www.memsic.com/.
6. Disponible sur notre site http://www igm.univ mlv.fr/PASNet/project_dlite.html.
7. Téléchargeable sur https://bec3.univ mlv.fr/learn.xhtml.

Table 4.1 Correspondances features et alphabet du FST (extrait)

Capteurs			
Feature	Abstraction	Messages	arguments
button	Bouton, interrupteur	push	
acceleration	Capteur de mouvement	accelerationChanged	X,Y,Z
gps	GPS	positionChanged	long.,lat.,alt.
joystick	Joystick	Click RightUp CenterDOWN Right ...	
light	Capteur de luminosité	lightChanged	X
Effecteurs			
Feature	Abstraction	Messages	arguments
led	led, luminaire, etc.	led	on off
notification	afficheur, écran, etc.	notify	chaîne à afficher
sms	téléphone	sms	numéro,message

ment, échanges inter-objets), et permet le déploiement de stratégies dans le but de construire des applications chorégraphiée. Le dialogue avec un objet D-LITeful est régi par notre langage *SALT*. La partie qui suit détaille le contenu et la syntaxe des différents formats SALT.

4.3 Formalisation du langage SALT

SALT (*Simple Application Logic description using Transducers for the Internet of Things*) a été conçu pour permettre à un programmeur de décrire l'ensemble de son application et les interactions entre les objets. La conception de SALT a été guidée par la volonté d'exprimer nos formes typiques de FST et de leur offrir toute l'expressivité nécessaire à la réalisation d'applications Internet des objets les plus adaptées. Tout objet disposant de notre framework D-LITe (nous utilisons parfois l'adjectif *D-LITeful*) comprend le langage SALT.

4.3.1 Besoins couverts par le langage

Le langage SALT est utilisé lors des 4 opérations typiques de notre approche concernant l'Internet des objets :

découverte des caractéristiques de l'objet ;
déploiement de la stratégie à exécuter ;
échanges de messages entre objets ;
effacement complet avant reprogrammation.

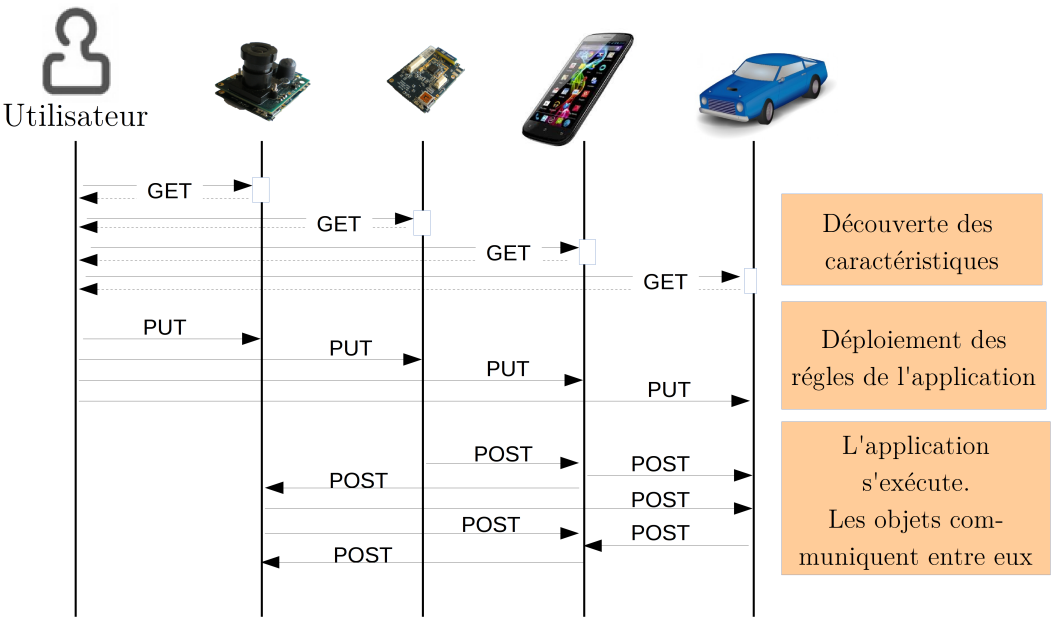

Figure 4.9 Dans notre vision, un utilisateur découvre les caractéristiques des objets qu'il utilise grâce à une requête. Il conçoit alors son application en tant que collaboration de briques logiques qu'il dépose sur chaque objet impliqué. Ensuite, les objets exécutent leur part du travail, et communiquent entre eux afin de se faire réagir les uns les autres.

Lorsqu'un nouvel objet est découvert, il est nécessaire d'en apprendre un peu plus sur ses caractéristiques (ses capacités de mesures, ou d'actions, etc.) et de découvrir l'abstraction matérielle utilisable. Le langage SALT classifie les différentes caractéristiques des objets sous le nom de "*features*". La liste de ces *features* (voir Table 4.1) est transmise lors de la découverte de l'objet (voir figure 4.9). Elles donnent au programmeur la liste des messages abstraits qu'il pourra invoquer dans son FST. Ainsi, un objet disposant de la feature *button* et *led* indique par là qu'il supporte les abstractions *push* (mot de l'alphabet du FST pour l'appui sur le bouton) et *led* (*led on* et *led off*, en sortie du FST, en déclenchent physiquement l'allumage et l'extinction).

La partie la plus riche de SALT concerne la description de la logique à exécuter. Conceptuellement, cette description concerne le transducteur, agrémentée de la liste des objets à l'écoute de la sortie de ce transducteur. La liste contient l'adresse de chacun des objets destinataires des messages. Au stade actuel d'utilisation de la plateforme, nous avons choisi d'abonner les nœuds à l'ensemble des messages émis, sans restriction. L'attrait que pourrait avoir des abonnements sélectifs entraînerait un surcoût en terme de description, et serait pénalisant notamment à cause de la taille limitée de la mémoire dont disposent certains nœuds (moins de 10 Ko pour les TelosB par exemple). Le transducteur est transmis au moyen d'un nombre réduit d'informations : l'état initial et les transitions uniquement (chacune composée de l'état de départ, message entrant, message sortant et état en sortie). Toutes

les informations nécessaires sont reconstituées à l'arrivée (liste des états, alphabet d'entrée et de sortie, états finaux si il y en a).

Les éléments de l'alphabet d'entrée et de sortie de SALT sont des messages (des chaînes de caractères) dont la sémantique a été améliorée. Alors qu'il n'y a jamais plus d'un seul message en entrée, voire aucun, lorsqu'on utilise l'élément vide ε (voir 4.1.3), il est possible d'utiliser de 0 à n éléments en sortie, et donc d'émettre plusieurs messages lors de la transition. Le détail des ajouts sémantiques et les différentes formalisations sont décrits dans les paragraphes qui suivent.

4.3.2 Extensions sémantiques et logiques dans SALT

Pour tenir ses promesses, SALT doit offrir l'expressivité nécessaire à la réalisation d'applications Internet des objets. Les formes *chorégraphiées* et *orchestrées* de ces applications doivent rendre les mêmes services. A. Barros et al. listent [28] 13 cas d'utilisation qu'elles doivent être en mesure de gérer. Ces 13 cas sont organisés en 3 grandes familles :

les *dialogues* (échanges entre nœuds pris deux à deux : envoi, réception, question/réponse...) ;

les *interactions multiples* (échanges entre plusieurs nœuds, avec sélection de réponses, dénombrement de réponses, bornage dans le temps...) ;

les *délégations* (charger un nœud d'en contacter un autre).

Les premiers cas sont assez simples à réaliser, s'agissant principalement de la possibilité d'envoyer ou de recevoir des messages entre deux partenaires, tâche dont s'acquitte aisément la *chorégraphie*. La deuxième famille (*interactions multiples*) présente des cas plus complexes d'échanges à plusieurs qui nécessitent, parfois, des prises de décisions liées, soit à un bornage dans le temps, soit à un nombre de réponses, voire les deux ensemble. Les auteurs terminent leur classification sur une gestion des redirections, par des systèmes de *délégation*. Si SALT sait exprimer cette organisation, il le fait de façon statique, a priori (l'utilisateur construit à la conception les liens utiles entre objets).

Les FST décrits avec SALT peuvent gérer dialogues et interactions multiples grâce à leur expressivité étendue :

Tout d'abord, SALT permet de piloter un **pseudo processeur** pour y créer des variables, stocker des valeurs, effectuer des opérations et tester leur contenu. Ce premier apport résout notamment les problèmes de comptage d'événements, mais ouvre aussi la porte à tout type d'opérations.

Ensuite, l'ajout d'un mécanisme de **timer** dans la machine virtuelle règle les problèmes liés au bornage dans le temps. Un timer peut être déclenché par un message sortant. Arrivé à échéance, il générera un message entrant "*timer*". Le programmeur peut alors déclencher des transitions à sa réception.

Enfin, chaque entrée ou sortie (les alphabets) peuvent contenir des **arguments**. Un élément de base peut par exemple être "*send,hello*", "*timer,1*" ou "*=,i,5*". Ces exemples de messages sortants indiquent pour le premier

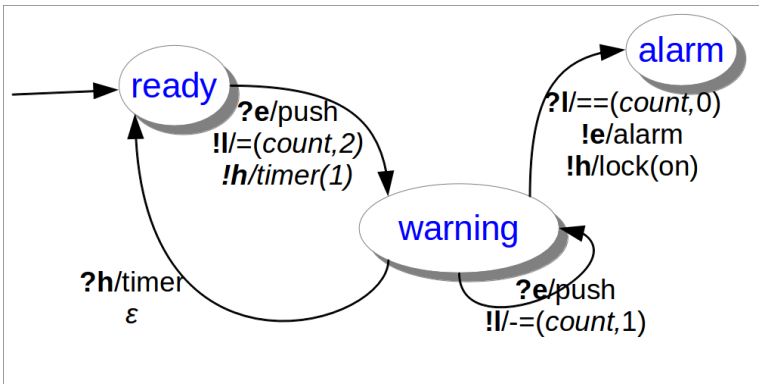

Figure 4.10 Un FST pour bloquer une serrure électronique dessiné avec notre outil de développement.

Table 4.2 Source SALT du FST de la figure 4.10

De	Entrée	Sortie	Vers
ready	?**e**/push	!**l**/=(count,1) !**h**/time(1)	*warning*
warning	?**e**/push	!**l**/-=(count,1)	*warning*
warning	?**h**/timer		*ready*
warning	?**l**/==(count,0)	!**e**/alarm !**h**/lock(on)	*alarm*

d'envoyer un message (*hello*), pour le second définir un timer d'une seconde, et pour le dernier la création d'une variable i de valeur *5*.

Le FST figure 4.10 et la version textuelle Table 4.2 donne un aperçu des possibilités. Pour chaque transition, le message entrant est symbolisé par un point d'interrogation, et les sorties avec un point d'exclamation. Une lettre indique le type de message (**e** pour *External*, **h** pour *Hardware* et **l** pour *Logic*), suivi ensuite du message complet. Partant de l'état "*ready*", une transition réagit à un message externe "*push*", en créant une variable *count* à 2, et en armant un *timer* d'une seconde. Arrivé à l'état "*warning*", si le *timer* se déclenche, une transition revient à "*ready*". Par contre, si un autre "*push*"est reçu, le compteur *count* est décrémenté. Lorsque sa valeur atteint zéro, le transducteur passe à l'état "*alarm*", émet un message "*alarm*" et déclenche le blocage matériel par l'effecteur. Ce FST permet de bloquer ce verrou si celui-ci reçoit trois messages "*push*" en moins d'une seconde.

Grâce à son alphabet sémantique étendu, les FST exprimés en SALT permettent d'expliquer à un objet qu'il doit compter le nombre d'événements reçus dans les 5 secondes, pour envoyer ce résultat à un autre objet, ou construire des mécanismes d'alertes de seuil, comparer des valeurs, etc. La sémantique de SALT autorise l'écriture d'applications plus riches que les simples interactions primaires, telles l'asservissement d'une led à un bouton ou le triple va-et-vient, réalisées avec les premières versions du langage.

L'organisation des éléments à l'œuvre dans SALT est décrite dans la Table 4.3 qui détaille le format d'une transition. La partie logique, qui orne l'alphabet d'une

Table *4.3* La structure des transitions dans le langage SALT

Formalisation des transitions : **État Entrée Sortie(s) ... États**	
État **xxxx**	Un simple mot **xxxx** qui indique la sémantique du raisonnement du programmeur. Aucune incidence particulière, aucun effet. Il s'agit simplement de nommer l'état courant dans la logique de l'algorithme.
Entrée **?x/yyyy**	Le **point d'interrogation**, suivi par **e**, **l** ou **h** (pour *externe*, *logique* ou *hardware*). Le prédicat (**yyyy**) peut revêtir plusieurs formes. Dans le cas des messages externes, il s'agit de la chaîne attendue. Si c'est un message logique, c'est un prédicat concernant une variable (un test). Sinon, il s'agit d'un message hardware exploitant les capacités des capteurs du nœud (tel que décrit au moment de la découverte du nœud).
Sortie **!x/yyyy**	Un **point d'exclamation**, suivi de **e**, **l** ou **h**. Un message externe est envoyé à tous les abonnés, un prédicat logique crée ou altère une variable dans l'objet, tandis qu'un message hardware déclenche une action. Il peut être paramétrée, par exemple pour définir l'action (*led(on)* ou *led(off)*) ou décrire l'opération logique(*=(nb,5)* ou *+=(count,2)*).
...	Une transition peut avoir de multiples messages en sortie(à destination du hardware, des abonnés ou pour les opérations sur les variables)
État **zzzz**	Un simple mot, décrivant l'état atteint par la transition dans le raisonnement du programmeur.

Table *4.4* Langage SALT : prédicats de gestion des variables

En entrée prédicats de (*test des variables*)	
?l/==(var,valeur)	Vérifie l'égalité de "*var*" et "*valeur*". Si elle s'avère, la transition est déclenchée. Les opérateurs acceptés sont < > <= >= et **!=**.
En sortie prédicats de (*définition ou de calcul de variables*)	
!l/=(var,valeur)	Donne la "*valeur*" à "*var*". Cette variable est crée si elle n'existait pas.
!l/+=(var,valeur)	Ajoute "*valeur*" à "*var*". *valeur* peut être une autre variable. Le résultat est stocké dans "*var*". 4 opérations sont gérées : (+=,-=, *= et /=)

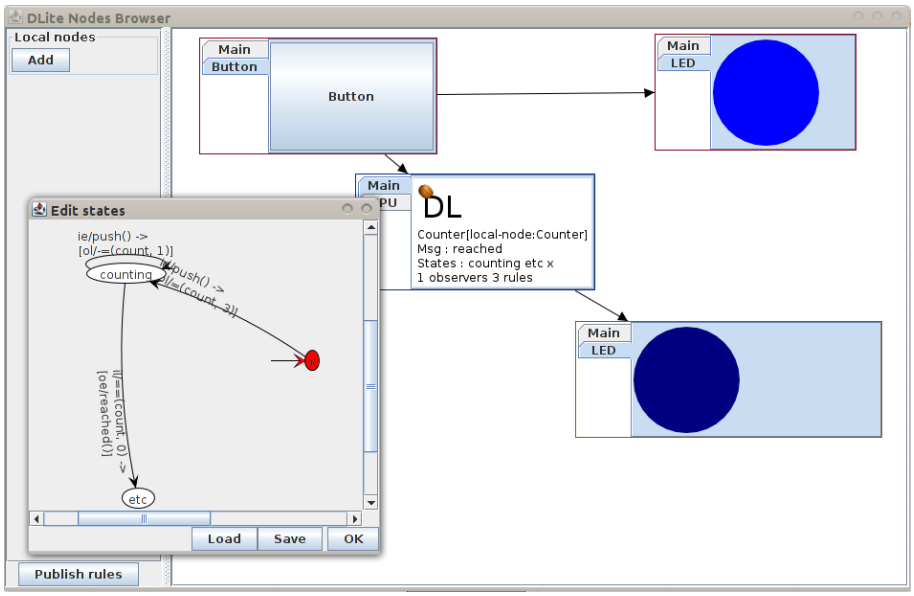

FIGURE *4.11* Dans l'outil de développement fourni avec SALT, le programmeur place les objets recensés ou virtuels(add) sur l'espace de développement pour les lier entre eux (flèches) et décrire la logique à exécuter. Ici, un bouton, un compteur et deux leds collaborent. L'algorithme déployé sur le compteur (visible dans la fenêtre surgissante) envoie le message "reached" à la led si le bouton a été utilisé 4 fois (voir le FST).

sémantique plus élaborée, est décrite dans la Table 4.4. L'utilisation de *prédicats*[8] plutôt que de simples chaînes de caractères améliore l'alphabet d'entrée et de sortie. Comme dans le monde des bases de données, ceux-ci permettent d'exprimer des contraintes sur des variables qui peuvent ou non être vérifiées. Tout en respectant le principe des transducteurs (il s'agit bien d'une entrée ou d'une sortie), les prédicats apporte une plus grande valeur sémantique aux alphabets.

4.3.3 Formalisation visuelle de SALT

Afin d'en démontrer la faisabilité, nous avons programmé sur différents objets le framework *D-LITe* (hébergeant l'interpréteur chargé d'exécuter les FST exprimés en *SALT*, et tous les services REST pour le déploiement de cette *stratégie*) ainsi que différents éditeurs pour faciliter la création des applications.

L'outil présenté figure 4.11[9] détecte réellement les objets de l'utilisateur, et peut aussi ne travailler qu'avec des objets virtuels (dans le but d'écrire génériquement du code). Dans les deux cas, les objets disponibles apparaissent sur le coté gauche de l'interface graphique. Le programmeur pourra ensuite les déposer sur son espace de création d'applications afin de les intégrer dans sa logique. Chaque objet peut être programmé puis intégré en tant qu'élément de l'application chorégraphiée. Le

8. En sémantique, un prédicat décrit une propriété ou une relation. Le terme est ici emprunté au monde des bases de données.

9. Disponible sur notre site `http://igm.univ mlv.fr/PASNet/SALT/`.

```
<node state="initial">
    <rules>
    [...]
        <rule>
            <input state="x">
                <message name="push" category="external" />
            </input>
            <output state="counting">
                <message name="=" category="logical">
                    <param>count</param>
                    <param>3</param>
                </message>
            </output>
        </rule>
        <rule>
            <input state="counting">
                <message name="push" category="external"/>
            </input>
            <output state="counting">
                <message name="-=" category="logical">
                    <param>count</param>
                    <param>1</param>
                </message>
            </output>
        </rule>
        <rule>
            <input state="counting">
                <message name="==" category="logical"/>
                    <param>count</param>
                    <param>0</param>
            </input>
            <output state="etc">
                <message name="reached" category="external" />
            </output>
        </rule>
    [...]
    </rules>
</node>
```

Figure 4.12 Le FST exprimé en SALT peut être enregistré sous un format XML. Cela lui permettra d'être échangé entre utilisateurs, et stocké dans notre base de données de briques logicielles, présentée dans le chapitre suivant

programmeur lie ses objets les uns aux autres, selon les interactions qu'il désire déclencher. Ces liens sont symbolisés par des flèches orientées dans le sens du contrôle.

Pour la partie programmation (création du *FST*), l'outil permet la création des états, et des transitions entre ces états. Le programmeur saisit ses messages entrant et sortants. Chaque message doit être catégorisé (*External*, *Hardware* ou *Logic*, voir 4.1.5) dans la boite de dialogue. Les messages externes auxquels ce nœud peut réagir apparaissent (par recensement automatique des sorties des FST de tous les objets reliés), mais sans contrainte d'utilisation. Les différents mots réservés à l'accès au matériel (*l'abstraction matériel*) sont automatiquement proposés (grâce aux *features* de l'objet obtenues par la commande GET, figure 4.9 et Table 4.1). Pour la partie logique, les messages entrants sont restreints aux tests de contenu de variables, tandis que les déclarations de variables et affections n'existent qu'en message sortant, respectant ainsi notre formalisation du langage Table 4.4. Le FST résultant est dessiné (figure 4.11).

Un fois tous les FST décrits et les liens entre objets créés, il devient possible de tester le bon fonctionnement de l'ensemble. Chaque objet symbolisé à l'écran dispose d'une interface le simulant (un bouton cliquable pour un interrupteur, une pseudo led pour un effecteur de type light, etc. figure 4.11). L'application s'exécute, et le programmeur/utilisateur peut la tester et la corriger si nécessaire. Pour la version interagissante avec les objets réels, le bouton "*publish rules*" déploie réellement les FST sur chacun des objets, qui se mettent alors à interagir.

4.3.4 Format XML de SALT

Afin d'organiser la persistance de la description de la logique créée par l'utilisateur, SALT propose un format XML. Le FST est linéarisé pour son stockage dans une base de données ou un fichier. Cette opération permet la sauvegarde, la restauration et le partage d'applications.

La figure 4.12 montre le code XML d'un FST qui compte 4 réceptions d'un message "*push*" puis déclenche l'émission d'un message "*reached*". La description du

FST est contenu dans un nœud racine *node* dont l'attribut obligatoire *initial* indique le nom de l'état initial. Dans *node*, on trouvera un nœud *rules*. Celui-ci contient une à une l'ensemble des transitions (*rule*). Chaque *rule* contient deux éléments correspondant à l'état en entrée (*input*) et en sortie (*output*) de la transition (indiqué par l'attribut *state*). *Input* peut contenir entre 0 et 1 *message*, alors qu'*Output* en accepte de 0 à n. Chaque message contient deux attributs obligatoires :

Le nom du message (*name*), qui sera comparé, pour un *input*, avec ce qui entre dans le FST. Si les contenus correspondent, la transition est déclenchée. Dans un *output*, c'est le message qui sera émis en sortie du FST.

Sa catégorie (*category*) qui indique à l'analyseur le flux utilisé, soit avec l'extérieur (méthode POST et message *external*), soit avec le matériel (message *hardware*, géré par la couche d'abstraction matérielle), soit au niveau logique.

Le message peut contenir de 0 à n paramètres *param*, contenant une chaîne de caractères, pour les besoins du programmeur. Il peut s'agir d'une précision sur un message ou d'arguments. Par exemple, le message *led* de la catégorie *hardware* accepte un paramètre ('*on*' ou '*off*') pour la prise en charge de l'opération par la couche d'abstraction matérielle. Les valeurs/variables modifiées par les messages *logic* en sortie sont transmises en paramètres.

Puisque D-LITe obéit à une approche événement, la réception d'un message provenant d'un autre nœud ou des capteurs internes (via la couche d'abstraction matérielle) sont indifféremment considérés comme des stimuli entrants. Même chose en sortie, le FST est exécuté de façon standard, et les messages à destination de l'extérieur sont émis sur le réseau (POST vers tous les abonnés) tandis que les messages matériel sont soumis à la couche d'abstraction matérielle.

Seule la partie logique est traitée légèrement différemment. En sortie, *logic* permet de définir ou d'altérer des valeurs. Par contre, une entrée *logic* est particulière, car c'est à l'arrivée dans un état que l'analyseur vérifie si celui-ci dispose de transitions logiques. Si c'est le cas, les tests sont évalués immédiatement, et la nouvelle transition déclenchée lors d'un résultat positif. La figure 4.12 est extrait d'un FST SALT exprimé en XML. L'état initial est "*initial*" (pour lequel aucune transition n'apparaît dans l'extrait). Des transitions pour 3 états (*x*, *counting* et *etc*) y sont décrites. La réception d'un message *push* venant d'un autre nœud emmène le FST de l'état *x* à *counting*, tout en initialisant une variable *count* à 3. Dans *counting*, chaque réception de *push* décrémente cette variable. Lorsqu'elle arrive à 0 (après 4 *push* en tout), le FST passe à *etc* en émettant le message "*reached*" à tous les abonnés. Comme indiqué plus haut, le test $count == 0$ est vérifié à chaque arrivée dans l'état *counting*.

4.3.5 Format compressé de SALT

Lorsque des objets de type capteurs sont impliqués dans l'application, la verbosité et le poids des balises structurant la description XML du FST pose le même problème que SOAP face aux contraintes des réseaux WSAN [39]. SALT propose un format très compressé de la description du FST pour ce type de réseaux contraints.

Même si XML propose sa version compressée (EXI), nous avons construit notre propre solution afin de profiter des spécificités de SALT et obtenir un degré de compression plus important. Elle s'effectue lors du passage dans la passerelle d'accès au réseau de capteurs. Celle-ci supprime par exemple les noms des états pour les remplacer par des nombres. Les états du FST n'ont d'intérêt que pour le raisonnement du programmeur, puisqu'ils ne représentent que des étapes logiques. Les chaînes utilisées dans le FST (noms des commandes, des variables, arguments) sont regroupées en fin du message, et les transitions utilisent les index de leur position. Cela permet notamment de capitaliser l'appel des commandes internes (les messages *hardware*) plus fréquents. Les FST utilisés généralement sur les objets présents dans les réseaux contraints sont suffisamment simples pour être compressés efficacement au regard des capacités de transmission [10].

L'ensemble des solutions présentées jusqu'ici dans ce chapitre donne aux objets connectés à l'Internet la possibilité d'être intégrés dans des applications IoT. Pour peu qu'ils supportent le framework D-LITe, dont le portage n'est pas très complexe à réaliser, ils peuvent décrire leurs caractéristiques selon les *features* référencés. Ils deviennent aussi (re)programmables à volonté, et ce grâce à SALT, langage simple mais suffisamment expressif pour implémenter à la fois des structures logiques et des accès standardisés au matériel, via une couche d'abstraction. Enfin, l'abonnement des objets entre eux et la faculté d'échanger des messages ouvrent la voie vers de véritables applications distribuées, chorégraphies d'objets interagissants, le tout utilisant des protocoles standardisés et une approche REST.

La prochaine partie traite des éventuelles failles qui peuvent apparaître dans de telles chorégraphies, et met à disposition un outil pour les corriger et limiter leur impact.

4.4 Tolérance et correction d'erreurs dans les applications chorégraphiées

Pour peu qu'elle mette en jeu des réseaux peu fiables, une architecture basée sur notre framework D-LITe est, comme toutes les solutions chorégraphiées, sujette aux erreurs. A l'inverse d'une orchestration dont le noeud central contrôle l'intégrité et est en mesure de gérer les dysfonctionnements, la distribution à l'œuvre dans les chorégraphies rend plausible l'apparition d'incohérences. Or, une part non négligeable de l'Internet des objets met en jeu des WSAN. Afin de limiter les impacts de la fiabilité limitée de ces derniers, nous proposons, dans la partie suivante, des mécanismes de vérification et de corrections distribuées adaptés à l'IoT.

10. La majorité des FST que nous avons conçus peuvent être compressés à une taille inférieure à 100 octets, et le code complet pour chaque objet transmis en un seul message.

4.4.1 Impacts de l'architecture applicative chorégraphiée dans les WSAN non fiables

La distribution du traitement est sujette à un important handicap lié à l'absence de point central de contrôle : l'ensemble peut se déstabiliser. Même si certaines des approches du WSAN, réseau réputé peu fiable, disposent de méthodes habituellement capables de compenser ces imperfections, celles-ci (la redondance par exemple) deviennent moins pertinentes dans le cadre de son utilisation pour l'IoT. Dans un réseau orienté Internet des objets, la diversité des composants croît tandis que le nombre d'exemplaires de chaque type diminue. Et même si, en *domotique* par exemple, les interrupteurs, lampes, capteurs, etc. sont parfois nombreux au sein d'un même environnement pour un unique utilisateur, chacun a bien un rôle distinct en dépit des redondances, et sera considéré comme unique. Dans le domaine des villes intelligentes, une organisation (une mairie, un campus universitaire, une entreprise...) peut disposer de multiples capteurs de place de parking et de présence, de multiples caméras, éclairages, panneaux d'affichage, feux de signalisation, etc. Cependant, là encore, les informations captées ne se recoupent pas. Bien que provenant de multiples sources de même nature, il n'est pas possible de réaliser des corrections ou des spéculations pour retrouver d'éventuelles informations manquantes (si par exemple un capteur de place de parking ne répond plus). Malgré les éventuels multiples exemplaires, chaque objet de l'Internet des objets est souvent considéré comme exemplaire unique parce que son rôle ne peut être délégué.

Dans notre approche, nous avons privilégié une approche orientée événement plutôt que données dans une organisation distribuée du traitement. Avec la chorégraphie qui résulte de ces choix, la moindre erreur réseau peut avoir des conséquences importantes.

4.4.2 Caractéristiques des applications IoT

Les applications IoT diffèrent de celles des réseaux de capteurs traditionnelles en plusieurs points :

Elles mettent en jeu d'*autres types de réseaux* aux propriétés variées.

Les réseaux de capteurs traitent souvent un *nombre limité de caractéristiques de l'environnement* en de multiples points avec des capteurs identiques, alors que les applications IoT manipulent de *nombreuses caractéristiques*, chacune d'elles grâce à un nombre restreint d'objets.

Alors que les applications de réseaux de capteurs sont plutôt orientées données, les applications IoT intègrent une dimension comportementale et, de ce fait, sont plus riches en traitements.

Les applications IoT recèlent des *sous-ensembles d'actions et réactions* (des comportements), qui peuvent être eux-même cohérents et autonomes, et qui parfois bouclent à l'infini.

Les erreurs des applications IoT sont en général causées par des messages perdus. Cependant, notre voulons prendre en charge un spectre plus large de dysfonc-

tionnements, liés au matériel, au logiciel, voire même à l'utilisateur. Effectivement, des solutions existent pour pallier le problème des messages perdus, l'ajout d'un mécanisme de contrôle par acquittement par exemple. Mais ici, notre objectif est d'obtenir une resynchronisation plus générale, afin d'améliorer la qualité globale de l'expérience utilisateur. Que les erreurs proviennent des défauts d'un réseau, d'un protocole ou d'une incohérence logicielle n'a finalement que peu d'importance.

De par la nature des applications concernées, et de leur mode de fonctionnement, certaines erreurs n'ont que peu d'incidences. Sur un algorithme "*bascule (flip-flop)*" courant (comme un interrupteur par exemple), la perte d'un message sera corrigée dès la réception du message suivant. Aussi, ce type d'algorithmes s'auto-synchronise, et ce d'autant plus facilement que sa fréquence est élevée et son nombre de pas limité. Par contre, d'autres algorithmes (comptage par exemple) ne s'auto-stabilise jamais, et la dérive engendrée par des erreurs s'accumule au fil du temps.

Les conséquences des erreurs pour les applications chorégraphiées sont d'importances très variées, parfois anodines car auto-correctrices, parfois jamais résolues. L'orientation fortement distribuée de notre vision des applications IoT et les particularités de leurs usages forment des structures logiques particulières. La construction d'un FST centralisé décrivant l'ensemble (combinaison des FST déployés) [122], qui semble plausible à première vue pour appréhender le comportement global de l'application et y insérer ensuite des mécanismes de vérification, s'avère rapidement trop complexe, et son analyse hors d'atteinte.

La fabrication de multiples arbres de vérification autonomes, indépendants et limités en taille va nous offrir un mode d'expression plus adapté aux besoins. L'ajout d'un mécanisme dans SALT permet au programmeur de disséminer des points de re-synchronisation des différents éléments afin de stabiliser son application, selon ses besoins et pour tout type d'erreurs. Ce dispositif est décrit dans les paragraphes suivants : sous-ensemble de nœuds soumis à vérification, invocation dans SALT, et algorithmes utilisés.

4.4.3 Surcouche de points de contrôle

Le concept est basé sur des vérifications ponctuelles à la discrétion du programmeur. Créateur de l'application, il connaît des combinaisons d'états stables d'objets. Sur l'ensemble des combinaisons possibles dans une chorégraphie, certaines sont remarquables. En domotique par exemple, on peut considérer que le verrouillage de la porte d'entrée doit correspondre à l'extinction de l'ensemble des lumières. Pour une application de ville intelligente gérant un parking, des vérifications peuvent être déclenchées par un système d'alarme incendie (au début, en cours ou en fin d'alarme) alors que d'autres ont pour déclencheur un état de la porte centrale. Tout décalage, qu'il soit du à une erreur réseau ou un oubli humain, peut être corrigé par une resynchronisation.

Même si les états stables de groupes d'objets peuvent être rares, il en existe au moins un : les états initiaux des objets, auxquels on peut avoir recours en cas de dysfonctionnement de l'application (réinitialisation). Selon les applications, le pro-

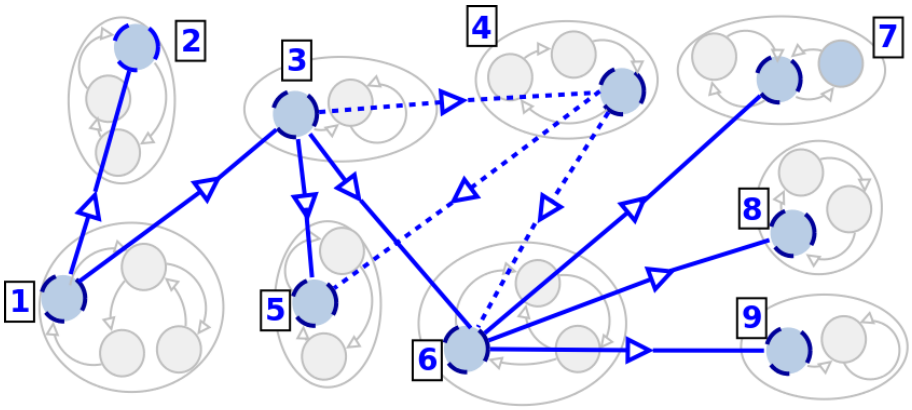

Figure *4.13* Surcouche de vérification : Il s'agit d'un arbre mettant en jeu un certain nombre de nœud de l'application. Un seul niveau est représenté ici, dans lequel l'objet 1, en tant que racine, est chargé du lancement de la vérification (lorsqu'il atteint l'état choisi). La vérification suit alors la cascade indiquée. A noter le "*OU*" utilisé entre les nœuds 3 et 4. D'autre part, le nœud 7 accepte deux états valides dans cette vérification.

grammeur peut en concevoir d'autres. Ces groupes d'états stables sont caractérisés par des dépendances spécifiques (d'où la construction d'un arbre dédié à chaque vérification) et un événement déclencheur (le passage à un état précis d'un objet). L'idée de construire une cascade au lieu d'une vérification globale par le nœud déclencheur tient au fait que certains objets disposent d'une mémoire limitée. La liste exhaustive de tous les nœuds à vérifier pourrait atteindre une taille rédhibitoire. D'autre part, des multiples flux émis par un même nœud peuvent être pénalisants (dans les WSAN par exemple). A contrario, le système de vérifications en cascade respecte l'orientation algorithmique distribuée de notre approche, et répartit la charge de travail.

Chaque vérification (qu'on identifiera par un numéro unique *niveau de vérification*) constitue un arbre spécifique, dont la racine de l'arbre est un état repéré du nœud déclencheur. La figure 4.13 montre une surcouche de points de vérification définie par le programmeur de l'application. Le nœud 1 est la racine de cet arbre. À chaque passage dans l'état entouré, le FST déclenche la demande de synchronisation. Les nœuds 2 et 3 reçoivent la demande, et y obéissent : ils se positionnent dans l'état entouré (si ils n'y sont pas déjà). Le nœud 2 n'a pas de descendant, le nœud 3 contacte ceux qui dépendent de lui. La cascade continue jusqu'à atteindre les nœuds extrèmes (7, 8 et 9).

Resynchroniser une application distribuée en utilisant un mécanisme lui-même distribué nécessite la possibilité d'exprimer des compositions logiques. En effet, le programmeur peut vouloir exprimer que si le nœud N a atteint l'état X, alors le nœud X doit être soit :

à l'état 1 ou 3 (**alternative interne**) ;
à l'état 1 et le nœud Y à l'état 2 (**ET entre nœuds**) ;
à l'état 1 ou le nœud Y à l'état 2 (**OU entre nœuds**).

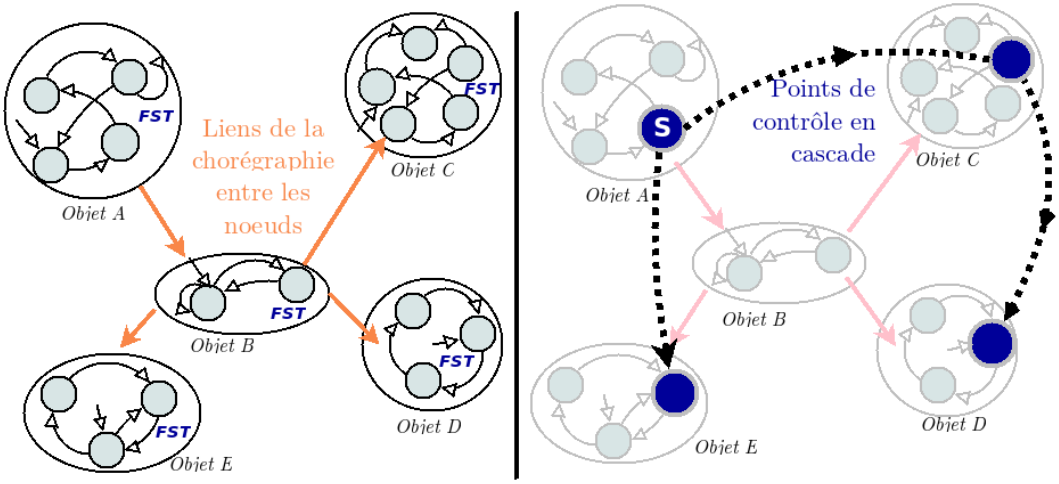

Figure 4.14 Dans D-LITe, une application est une chorégraphie de FST. Chaque élément suit sa propre logique. Afin d'améliorer la cohérence globale de l'application, le programmeur définit des combinaisons d'états qu'il considère comme étant stables. L'arrivée dans l'état S de l'objet A déclenche les vérifications, qui se propagent en cascade.

Sur la figure 4.13, le nœud 7 utilise l'**alternative interne**. Pour ce niveau de vérification, il accepte deux états valides. Le **ET entre nœuds** est facile à réaliser, c'est le cas normal. Un **OU entre nœuds** est visible entre les nœuds 3, 4 et 5. Si le nœud 3 est dans l'état attendu, alors le **OU** est directement vérifié, et la cascade passe aux nœuds 5 et 6 (4 n'a pas à être vérifié). Mais si 3 n'est pas dans un état valide, il faut alors vérifier l'autre membre du **OU** (le nœud 4). Seul ce dernier sera corrigé si nécessaire, puis la vérification continuera sur les nœuds 5 et 6.

Dans une vérification centralisée, les **OU** auraient été plus difficiles à réaliser car ils imposent un retour afin de savoir si on doit évaluer le second membre de l'expression logique, impliquant une communication synchrone et bidirectionnelle, discordante par rapport à notre approche globale *chorégraphiée*.

4.4.4 Commandes de resynchronisation

Après avoir défini ses vérifications (identifiées par leur niveau) et l'arbre sur lequel chacune agit, le programmeur doit les retranscrire dans SALT. A l'instar des assertions présentes dans certains langages comme le C, cette demande de vérification est une affirmation du programmeur, ne faisant pas progresser son raisonnement, et liée à l'état de cohérence du logiciel. En C, si une assertion est erronée, l'exécution du programme est définitivement interrompue. D'autres langages déclenchent des *exceptions* afin de traiter l'incohérence logique constatée.

Nous proposons un système d'assertions déclenchant la cascade de vérifications (figure 4.14). Deux tâches incombent alors au programmeur :

> Indiquer la commande "*check*" suivie du niveau de vérification sur le nœud racine de chaque arbre. Dans la figure 4.14, les transitions du FST de l'objet A qui amènent à l'état S invoquent toutes un "*check*".

Définir, pour chaque objet, la liste des états compatibles avec chaque niveau de vérifications. Aussi, dans la figure 4.14 devra-t-il indiquer aux objets E, C et D l'état correspondant à la vérification indiquée. À l'objet C, il indiquera de plus qu'il est chargé de la cascade vers D. Les objets E et D n'ont pas de descendant. Notre framework se charge de la correction éventuelle dès réception du message, et de la poursuite de la cascade.

L'expression du **OU** figure 4.13 se déclare, au niveau du noeud 3, par son état valide et sa cascade vers 5 et 6. Une cascade alternative pointe vers 4, et ne sera utilisée que si la demande de check trouve cet objet dans un état invalide. Au nœud 4, le programmeur stipule son état valide, et sa cascade vers 5 et 6. Cependant, cette vérification ne sera déclenchée que si 3 est dé-synchronisé au moment de sa vérification par 2. Dans le cas où 3 est valide au moment du *check*, 4 ne reçoit aucune demande de synchronisation.

La commande "*check*" et les listes d'états compatibles avec chaque niveau sont proposées dans le langage SALT. Ces informations sont déployées au même moment que les FST. Elles sont ensuite automatiquement prises en compte par chaque nœud D-LITe selon les algorithmes qui suivent.

4.4.5 Algorithme de resynchronisation

L'arbre utilisé par la surcouche de points de contrôle est unique. Il ne correspond pas systématiquement à la liste des nœuds abonnés. La racine (le nœud déclencheur du contrôle) peut différer selon les arbres.

Algorithme 1 : Déclenche une vérification à la fin d'une transition du FST

```
Données : ckps Tableau de ckp, currentState
Résultat : Lance un check si nécessaire
début
    i ← 1
    tant que chkp[i] ≠ null faire
        si chkp[i].state = currentState alors
            chkNumber ← random
            pour tous les ip dans chkp[i].targets faire
                sendChkMsg(ip, chkp[i].chkId, chkNumber)
            Retourne Vrai
        sinon
            i ← i + 1
    Retourne Faux
```

L'algorithme 1 gère le déclenchement de la vérification sur la racine d'un des arbres de vérification (figure 4.13). Cette routine est exécutée à chaque changement d'état d'un FST. Si l'état fait partie de ceux qui sont référencés comme racine d'une vérification (le programmeur a indiqué un "*check*"), l'algorithme génère un numéro aléatoire *chkNumber* qui sert à la signer (dans une cascade distribuée, une

erreur créant une boucle de vérifications deviendrait infinie. Ce numéro référencera le passage de cette vérification, chaque nœud n'y obéissant qu'une seule fois). Le *chkNumber* et le *niveau de cette vérification* (l'identifiant de cet arbre) sont envoyés à l'ensemble des nœuds référencés pour ce niveau (nœuds 2 et 3 de la figure 4.13, le nœud 1 étant racine).

La réception d'un message de vérification est traitée différemment. Le nœud fait partie de la cascade, et n'est pas à l'origine de cette vérification (il n'est pas racine). L'algorithme 2 décrit le procédé utilisé. Le message arrive comme un message externe, il contient donc un caractère spécial d'identification (un point d'interrogation). Tout d'abord, le nœud vérifie qu'il ne l'a pas déjà traité grâce au *chknumber* du message. S'il doit traiter, l'algorithme recherche dans son tableau des *niveaux de vérification* la liste des états valides *states* pour ce *niveau*. Si la comparaison est valide, un message contenant le *niveau* et le *chknumber* est transmis à tous les nœuds fils de celui-ci dans l'arbre de vérification *targets*. Si l'état ne correspond à aucun état valide, alors, s'il existe une liste alternative *altTargets* (une implémentation du **OU entre nœuds** comme sur la figure 4.13 pour les nœuds 3 et 4), l'état est inchangé et c'est la liste alternative qui est utilisée pour la cascade. Sinon, en l'absence de *altTargets*, on force aléatoirement le FST à un des états valides pour ce niveau de vérification, et on déclenche la cascade sur la liste normale (*target*).

4.5 Etude expérimentale du mécanisme de stabilisation

La base de notre proposition consiste à offrir des possibilités de resynchronisation distribuée intégrées dans le langage de description de la logique applicative. Le programmeur pourra insérer là où il le juge nécessaire des cascades de vérification et de resynchronisation de ses machines à états. Un arbitrage doit être fait entre fiabilité et charge du réseau, puisque la resynchronisation a un coût en terme de nombre de messages émis. L'expérience présentée ici a pour objectif d'évaluer l'impact des erreurs sur une application sensible, et de présenter les liens entre l'augmentation du trafic et le gain en fiabilité apportés par le mécanisme correctif proposé.

4.5.1 Scénario de l'expérience

Le scénario choisi est typique d'applications qui ne tolèrent aucune erreur. Il s'agit d'un système de comptage, dans lequel un élément déclenche des événements. Afin de générer une grande sensibilité aux pannes, le comptage est organisé sous forme de chaîne. La figure 4.15 montre les deux groupes d'objets que nous avons utilisés et qui exécutent le même algorithme. Dans chaque groupe, un générateur émet des événements. Un premier nœud compteur (N_1) le capte, le compte et émet à son tour un message. Un second compteur N_2 compte ces messages, et émet lui-même à destination du dernier compteur N_3. Au final, on s'intéressera à la qualité de l'information détenue par ce dernier nœud. Il voit ses risques de dé-synchronisation augmenter à cause de sa position en fin de parcours. L'expérience utilise deux groupes pour à la fois disposer d'une référence et pour pousser le réseau à un niveau proche de

Algorithme 2 : Traitement de la réception d'un message de vérification

Données : *ckpRcvs* Tableau de *ckpRcv* (structure de stats[] (états valides), targets[] (nœuds cascades), altTargets[](liste alternative)) indexé par NiveauCheck, *currentState*, *LastChkNumber*

Résultat : Vérification de l'état, correction si nécessaire, et propagation

début

- **si** $RcvChkNumber \neq LastChkNumber$ **alors**
  - $LastChkNumber \leftarrow RcvChkNumber$
  - $i \leftarrow 1$ `/* Parcours des états valides pour ce NiveauCheck */`
  - **tant que** $ckpRcvs[RcvChkId].states[i] \neq null$ **faire**
    - **si** $ckpRcvs[RcvChkId].states[i] = currentState$ **alors**
    - `/* Cohérent : on cascade. */`
      - **pour tous les** *ip dans* $ckpRcvs[RcvChkId].targets[i]$ **faire**
        - sendChkMsg($ip, RcvChkId, chkNumber$)
      - Retourne Vrai
    - **sinon**
      - $i \leftarrow i + 1$

  - `/* Incohérence détectée */`
  - **si** $ckpRcvs[RcvChkId].altTargets = null$ **alors** `/* Pas de liste alternative */`
    - $newState \leftarrow 1 + (random() \bmod (i - 1))$
      `/* Choix aléatoire d'un état valide de NiveauCheck */`
    - changeFSTto($ckpRcvs[RcvChkId].states[newState]$)
    - **pour tous les** *ip dans* $ckpRcvs[RcvChkId].targets[newState]$ **faire** `/* Suite de la cascade selon ce nouvel état */`
      - sendChkMsg($ip, RcvChkId, chkNumber$)
    - Retourne Vrai
  - **sinon** `/* Traitement du OU, le FST reste en l'état */`
    - **pour tous les** *ip dans* $ckpRcv[RcvChkId].altTargets[]$ **faire**
      `/* Cascade du check sur la liste alternative */`
      - sendChkMsg($ip, RcvChkId, chkNumber$)
- **sinon** `/* Ce Check avait déjà été traité */`
  - Retourne Faux

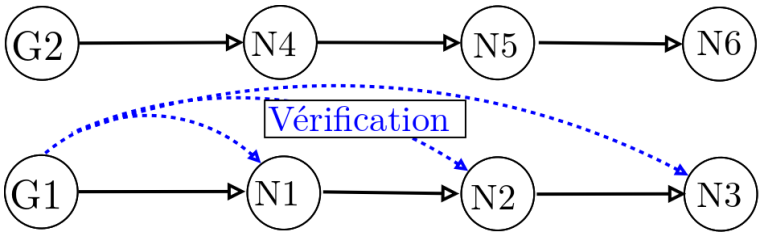

Figure *4.15* Notre expérience de resynchronisation utilise 2 groupes de 4 objets. Les objets G_x génèrent des messages et les nœuds N_x les comptent en cascade. Le premier groupe utilise le mécanisme de resynchronisation entre le nœud G et les nœuds N. Le second groupe permet de vérifier le décalage qui s'installe, et participe à saturer le réseau.

ses limites. Seul le premier groupe implémentera le mécanisme de re-synchronisation. L'expérience s'intéresse aux effets de cette resynchronisation, en faisant notamment varier la fréquence de son exécution.

D'autres paramètres ont été utilisés pour cette expérience. Tout d'abord, comme pour les expériences du chapitre 3, l'émulateur/simulateur Cooja et notre banc de tests ont été mis à contribution pour ces expériences. Nous nous sommes aperçus que le banc de tests était plus sensible aux erreurs que Cooja. Cooja simule le réseau, et il n'y a pas d'impondérable exogène. Le banc de tests de notre laboratoire est soumis à l'environnement et baigne dans les émissions des réseaux WiFi voire d'autres expérimentations en cours utilisant 802.15.4. D'autre part, Cooja émule le comportement des objets, et on peut supposer qu'il le fait avec une meilleure réactivité que l'objet réel (fréquence du processeur par exemple). Toujours est-il que nous n'obtenons pratiquement aucune perte en bout de chaîne sur Cooja, alors que l'expérience de base génère un taux d'environ 10% de pertes sur le banc de tests (figure 4.17). Afin d'obtenir une plus large gamme de résultats, nous avons introduit un générateur d'erreurs dans notre programme pour déclencher des pertes de paquets. Nous ferons donc varier le taux d'erreur et la fréquence des vérifications afin de mesurer leurs impacts en termes de précision des résultats et d'augmentation du trafic. La précision des résultats est évaluée selon l'écart entre le nombre d'événements générés et la valeur mesurée.

L'expérience s'est déroulée de la façon suivante : les algorithmes de comptage sont déployés sur chaque nœud, que ce soit dans l'émulateur/simulateur Cooja ou dans les nœuds du banc de tests. Une pause de sécurité de 10 secondes pour laisser l'ensemble des messages atteindre leur cible s'écoule avant de reprendre l'algorithme à son début. C'est lors de cette pause de 10 secondes que la demande de re-synchronisation est lancée. Chaque nœud affiche alors le nombre de messages qu'il a reçu, et se re-synchronise si nécessaire. Durant l'expérience, nous avons aussi fait varier le nombre de messages avant vérification. Lorsque ce n'est pas indiqué, la suite contient une synchronisation tous les 12 messages.

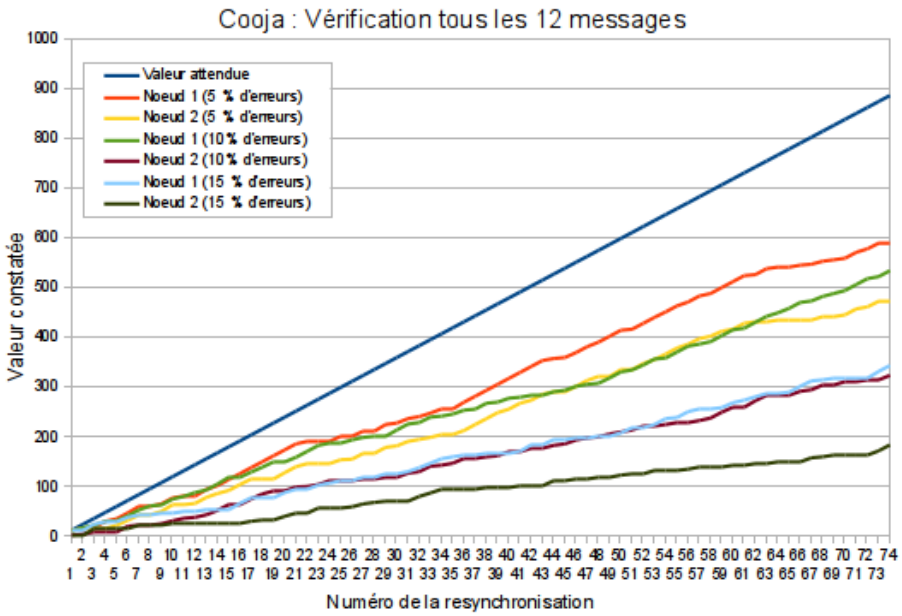

Figure *4.16* Sans le mécanisme de re-snchronisation, les valeurs mesurées s'éloignent de plus en plus de la réalité. Plus le nœud est loin dans la chaîne et plus le taux de pertes est élevé, plus le nombre de messages réellement reçus diminue.

4.5.2 Nécessité des corrections

La figure 4.16 recense les nombres de messages qui sont réellement parvenus aux nœuds selon la position de ceux-ci dans la chaîne et le taux d'erreur. Par rapport à la ligne droite qui représente le résultat attendu, les sommes obtenues montrent les impacts des imperfections du réseau selon les nœuds. Lorsque les algorithmes déployés ne sont pas auto-stabilisateurs, le mécanisme de correction s'avère nécessaire.

Dans notre application particulièrement sensible aux erreurs et soumises à des flux susceptibles d'en faire apparaître, les corrections sont inévitables pour maintenir les sommes calculées à des niveaux très proches de la réalité. La figure 4.17 permet de visualiser les corrections opérées par notre système. Lorsque nous n'introduisons aucune erreur, les demandes de synchronisation n'entraînent aucune correction sur Cooja. Mais sur notre banc de tests, la charge induite par notre expérience montre que, selon la position du nœud, de 10% à 40% des synchronisations détectent une erreur et aboutissent à une correction utile et efficace. La variation du taux d'erreur accroît proportionnellement les besoins de corrections, jusqu'à ce que celles-ci deviennent elles-mêmes non-fiables, les demandes de resynchronisation étant perdues à leur tour.

La figure 4.18 rend compte de la qualité de l'application en termes d'exactitude du calcul et du surcoût induit par les vérifications. L'exactitude est ici évaluée par rapport à l'écart de réponse de l'ensemble des nœuds. L'application est considérée exacte si et seulement si les 3 nœuds donnent la bonne réponse. Cependant la plupart

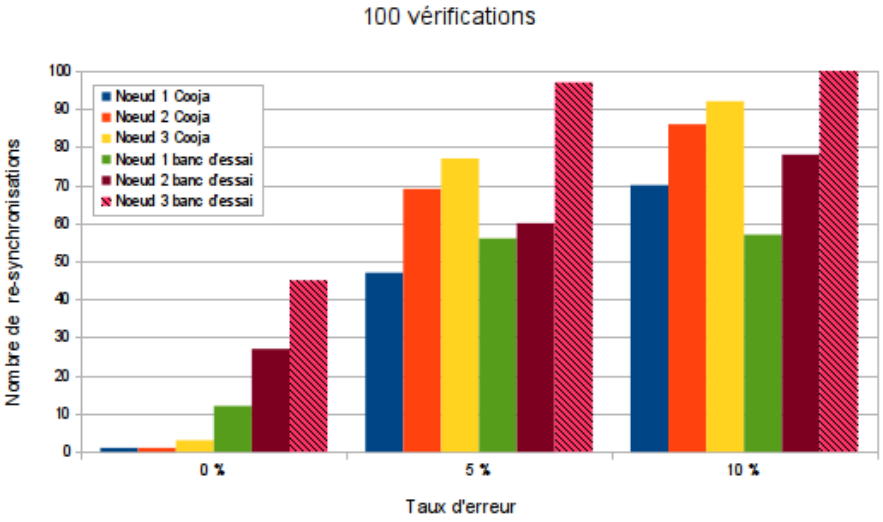

Figure 4.17 Afin de corriger l'application, nous lançons une vérification tous les 12 messages. Plus le taux d'erreur est élevé, plus la vérification est utile afin de rester au plus proche de la valeur correcte. Cependant, si le taux d'erreur devient trop important, les checks sont eux aussi perdus et l'application dérive de plus en plus. Le dernier objet de la chaîne (le nœud 3) est rapidement incohérent, particulièrement sur le banc de tests.

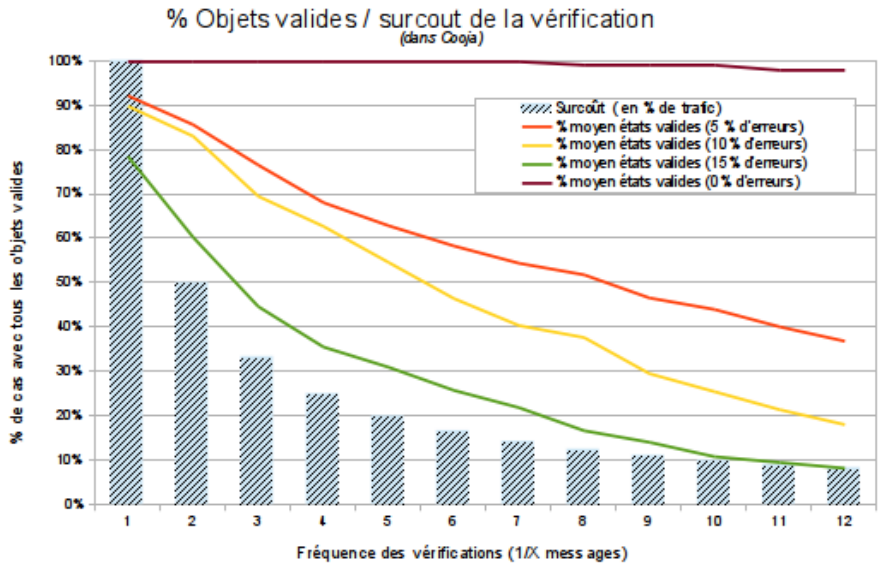

Figure 4.18 Les courbes représentent le pourcentage de cas dans lesquelles tous les nœuds ont les bonnes valeurs. Selon le taux d'erreur et la fréquence des vérifications (d'1 vérification par message à 1 pour 12 messages), la courbe donne la précision des valeurs de l'application. Pour obtenir un taux de réponses exactes proche de 70%, une vérification tous les 3 messages est nécessaire si le taux d'erreur est de 10%. Dans les 30% restants, la réponse obtenue cependant reste très proche de la bonne valeur. L'histogramme en bâtons trace le coût de la vérification.

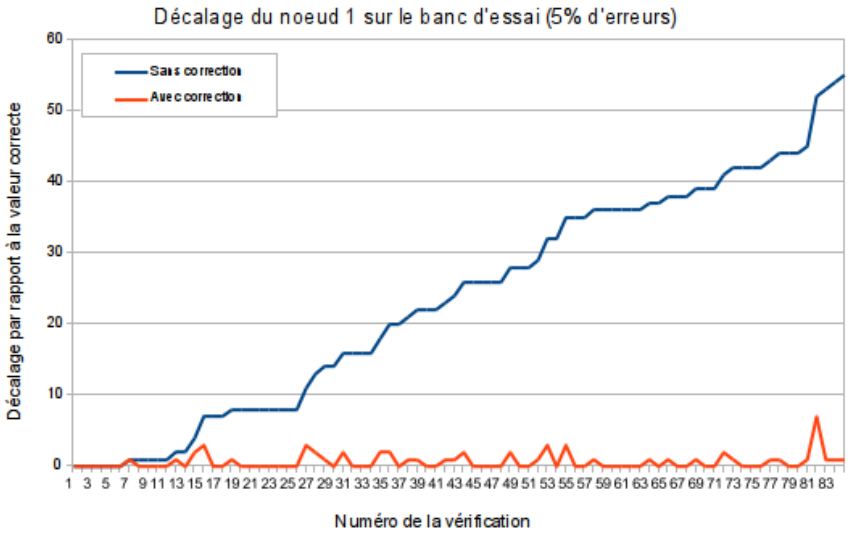

FIGURE *4.19* Grâce à la resynchronisation, l'algorithme déployé reste dans un intervalle d'erreur raisonnable. Sans ce mécanisme, l'application se décalerait de plus en plus, les résultats devenant incohérents.

des réponses dites "fausses" ne sont que faiblement décalées de la réalité, souvent de 1, et ce durant un laps de temps limité, puisque le mécanisme de synchronisation va corriger les écarts.

L'histogramme strié, en surimpression sur le graphique, indique le coût du trafic réseau du aux vérifications plus fréquentes. Réaliser une validation de chaque message entraîne un surcoût de 100%. Le programmeur doit arbitrer entre la précision et ce coût. Avec un taux d'erreur de 5%, notre application doit faire une vérification tous les 4 messages pour obtenir une précision de près de 70% de valeurs exactes sur tous les nœuds. Cependant, il faut noter que, dans les 30% restant, l'écart avec la bonne valeur reste limité. La synchronisation, envoyée ici tous les 4 messages, remet l'ensemble des nœuds en phase. Le mécanisme de resynchronisation contient l'écart possible dans un intervalle très restreint.

4.5.3 Contrôle des dérives dûes aux erreurs

L'apport de la resynchronisation apparaît sur les figures 4.19 et 4.20. Avec un taux d'erreur de 5%, les courbes donnent le décalage entre la valeur réelle et celle constatée par le nœud testé. La première courbe trace la dérive qui s'installe au fur et à mesure si aucune correction n'est réalisée, s'éloignant de la réalité. Au contraire, avec la correction activée, la dérive reste maîtrisée, l'application se maintient très près de la valeur juste. Sur la figure 4.20, les écarts se creusent, car ce nœud est dépendant du premier. Les erreurs s'ajoutent, mais l'efficacité de la re-synchronisation endigue l'écart dans des bornes acceptables. Afin de montrer la régulation des écarts que le système permet, la distribution de la figure 4.21 trace les niveaux des diffé-

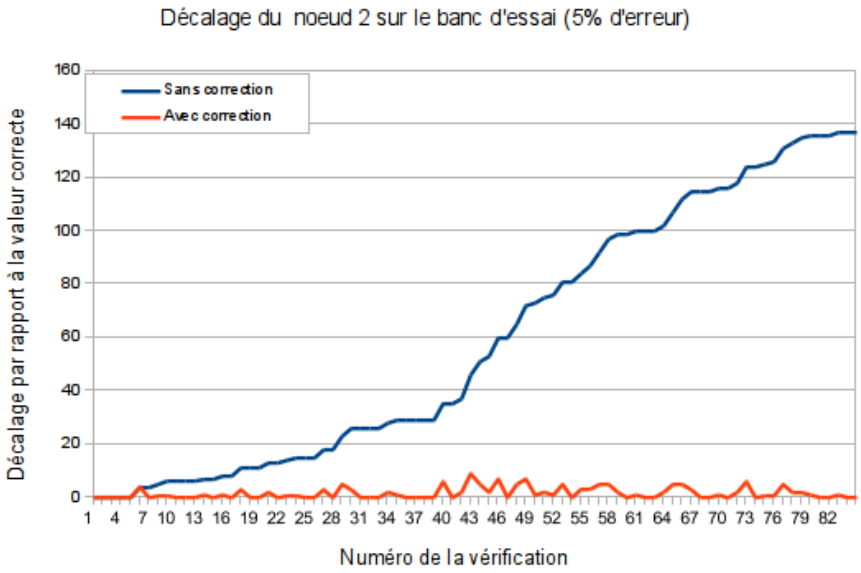

Figure 4.20 L'objet 2 dépend de l'objet 1. Les erreurs s'accumulent, et les corrections sont de plus en plus importantes.

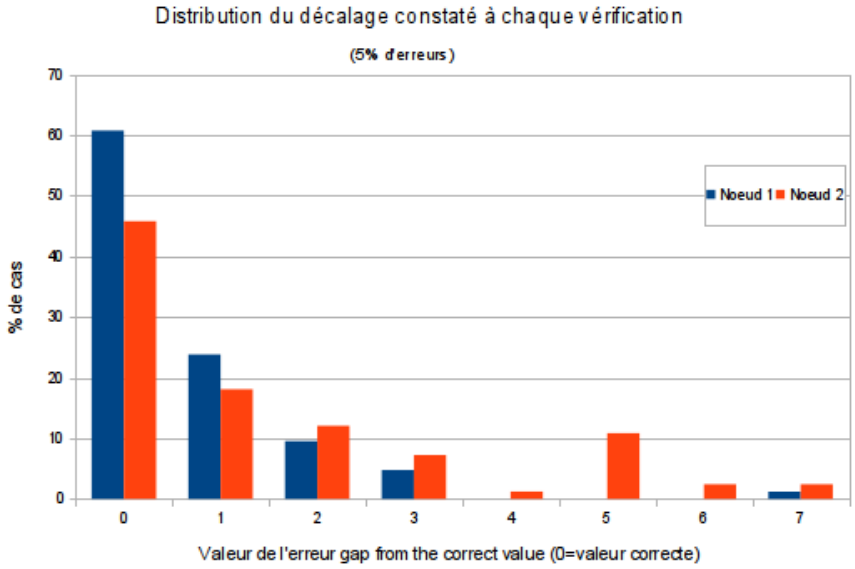

Figure 4.21 Cette distribution montre l'écart constaté par le mécanisme de correction proposé. De par les resynchronisations régulières, l'écart, quand il existe, demeure faible.

rentes dérives qui ont été constatés sur les nœuds 1 et 2 lors des expériences sur notre banc de tests avec un taux d'erreur généré de 5%. Dans près de la moitié des cas, même le nœud 2 ne présente aucun écart, et les erreurs du nœud 1 ne sont supérieures ou égale à 3 que dans moins de 10% des cas. L'application demeure fiable, la dérive restant limitée après plus de 1000 événements comptés.

4.6 Conclusion

Notre vision de l'Internet des objets nous amène à le considérer comme un assortiment d'actions et de réactions entre éléments hétéroclites, communiquant grâce à des événéments. Afin de construire une application capable de s'exécuter sur une large gamme d'objets, nous proposons l'utilisation d'une machine virtuelle. L'implémentation de notre solution sur divers matériel montre que la solution est réaliste. Grâce à une couche d'abstraction matérielle, notre solution D-LITe résout la difficulté liée à l'accès au matériel, et masque les différences entre des systèmes pourtant incompatibles à l'origine, qui deviennent alors interchangeables. La logique de chaque élément peut s'exprimer à l'aide du langage SALT que nous proposons, et qui est basé sur les transducteurs à états finis (FST). L'utilisation des FST masque là aussi les différences logicielles entre les éléments en jeu, et permet une expression plus universelle des rôles de chacun d'eux. L'utilisation de l'approche REST autorise le dépôt du programme sur chaque objet objet à distance, et les dotent une communication entre eux selon ce protocole largement validé et présent sur nombre de réseaux. Mais puisque la conception d'application distribuée est susceptible de ne pas avoir la même expressivité que des applications orchestrées, nous avons étendu les contenus des alphabets entrants et sortant des FST utilisés en étendant leur sémantique. Des prédicats décrivent les opérations logiques, et l'utilisation d'arguments rend notre langage SALT plus apte à concurrencer l'expressivité des applications orchestrées. Enfin, afin de pallier les incohérences qui peuvent survenir dans les applications chorégraphiées, notamment sur des réseaux non fiables comme les WSAN, nous proposons un mécanisme de re-synchronisation distribuée des automates, dans le but de contenir l'application dans une marge d'erreur acceptable, au regard de son impact en terme de surcoût de charge réseau.

Notre plateforme D-LITe fournit une base stable pour la construction des briques logiques pour l'Internet des objets. Nous allons maintenant nous intéresser, à un plus haut niveau, à la façon dont toute cette infrastructure peut devenir la fondation d'une architecture de construction simplifiée et agile d'applications.

Chapitre 5

Internet des objets : l'abstraction des échanges pour le développement agile

Sommaire

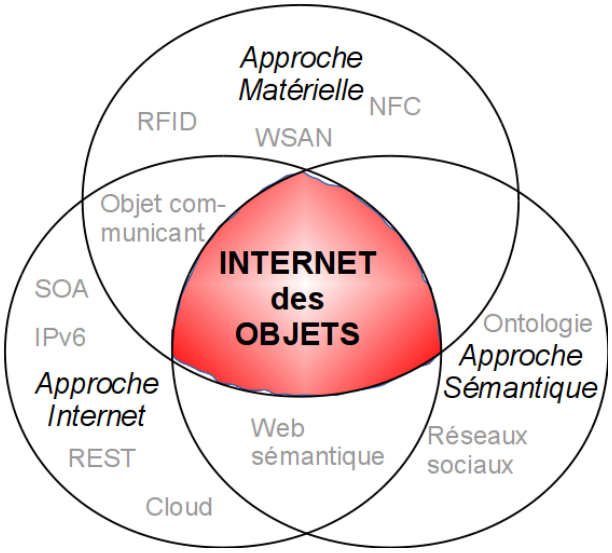

Figure 5.1 Ce schéma, adapté de la proposition de L. Atzori et al., présente l'Internet des objets comme la convergence de différentes approches et cultures technologiques, source de ses multiples définitions.

Les événements sont l'écume des choses.

Regards sur le monde actuel
Paul Valéry

L'Internet des objets fait l'objet de multiples définitions, selon la culture technologique de l'auteur qui la propose et l'orientation de ses travaux. L. Atzori et al. [25] proposent une typologie intégrant les différentes communautés de recherche qui composent l'Internet des objets, à partir des différentes approches utilisées. La figure 5.1 référence les principaux domaines qui s'inter-pénètrent dans ce que les auteurs nomment le "*Internet of Things paradigm*".

Les chapitres précédents traitent de l'Internet des objets selon les perspectives liées aux réseaux de capteurs et à l'Internet. Notre axe de travail s'est concentré jusqu'alors à bâtir une architecture qui soit à la fois la plus adaptée aux contraintes élevées des WSAN (car il s'agit du plus petit dénominateur commun de l'ensemble des réseaux qui s'intègrent dans l'IoT[90]) tout en respectant au mieux les standards utilisés sur l'Internet (afin que les échanges entre les différents protocoles applicatifs de ces multiples réseaux soient les plus universels).

Ce chapitre va présenter nos travaux sur un Internet des objets de plus haut niveau, plus "*agile*" et plus orienté sémantique. Les méthodes de développement "*agiles*" (voir le manifesto [29] de Beck et al.) sont basées sur une réduction des intermédiaires entre les programmeurs et les utilisateurs, un cycle de développement plus court avec des versions intermédiaires constamment en évolution, et une amélioration de la réactivité par une plus grande implication de ces derniers.

Pour rendre notre plafeforme "*agile*", nous allons introduire un niveau d'abstraction supérieur, en considérant maintenant les briques applicatives écrites en *SALT*

comme étant elles-mêmes des éléments interchangeables du comportement des objets. Ces briques applicatives deviendront les éléments de base de notre réflexion. Grâce à l'abstraction matérielle et à la définition d'un langage universel, *D-LITe* autorise la description standardisée du comportement de chaque objet à l'intérieur d'une application. BeC^3 (**Be**haviour **C**rowd **C**entric **C**omposition), notre solution de composition présentée dans ce chapitre, s'appuie sur l'abstraction comportementale, garantissant la conformité des applications combinant des fragments d'algorithmes classifiés et créés *par* et *pour* les utilisateurs. Basé sur une démarche de création participative et d'échanges (*Crowd-sourcing*), BeC^3 introduit un ensemble de solutions pour interconnecter, organiser et assembler des composants logiciels "Internet des objets" partagés entre utilisateurs, dans le but de bâtir des applications.

Ce chapitre s'intéresse tout d'abord aux différentes abstractions présentes dans BeC^3, et les gains qu'elles offrent. Ensuite, nous verrons comment notre système se nourrit des apports du Crowd Centric. Nous présenterons un rapide descriptif de la solution que nous avons élaborée, pour ensuite introduire la modélisation des échanges et les mécanismes de vérification interne. Un cas d'utilisation viendra clore ce chapitre.

5.1 Les apports et les limites de l'abstraction matérielle

L'abstraction matérielle proposée au programmeur par *D-LITe* simplifie l'écriture des briques logiques à exécuter sur chaque objet. Cependant, une proposition d'architecture pour l'Internet des objets doit faciliter la réalisation d'applications basées sur la collaboration entre objets à disposition de l'utilisateur/programmeur [68]. La question à laquelle nous allons essayer de répondre porte sur l'atteinte de l'objectif fixé : notre solution est-elle suffisante pour décrire cette collaboration et produire des interactions fiables ?

5.1.1 L'indépendance du code par l'abstraction matérielle

L'abstraction matérielle est une démarche bien connue. En introduisant un intermédiaire logiciel chargé de la gestion spécifique du matériel d'un coté, et qui expose une interface normalisée de l'autre, on délègue les particularités à un médiateur spécialisé. L'abstraction augmente la portabilité du code créé. En effet, sur un matériel différent, l'intermédiaire, s'il existe, offrira les mêmes méthodes génériques d'accès du point de vue de l'extérieur. C'est le cas par exemple d'un système d'exploitation dont l'objectif est d'être portable, universel et qui dialogue avec chaque matériel via un pilote adapté. L'appel aux fonctionnalités attendues est normalisé, documenté, et sa mise en œuvre à la charge du pilote. C'est encore le cas avec certains langages comme *Java* (ou le nôtre, *SALT*) qui proposent une machine entièrement virtuelle, l'abstraction étant gérée sur la machine hôte. Le code applicatif devient générique, et peut s'exécuter partout où l'abstraction est résolue.

5.1.2 Les besoins de l'Internet des objets

L'Internet des objets consiste à faire interagir un grand nombre d'éléments, chacun très spécialisé, dans le but d'obtenir une combinaison complexe d'actions et de réactions. Plus cette combinaison est riche, plus l'apport de l'application en termes de services rendus à l'utilisateur est important. Son expérience globale est liée à l'intégration transparente et sans effort des outils qui l'entourent ou auxquels il a accès à distance. L'Internet des objets doit permettre précisément la mise à disposition de l'extension numérique de l'environnement de l'utilisateur, le libérant des problématiques de configuration, d'interprétation et de connexions entre éléments.

Atteindre cet objectif d'informatique "*ubiquitaire*" (c'est-à-dire dans lequel est plongé l'utilisateur, selon la définition de M. Weiser [133]), omniprésente et invisible ("*pervasive*" présentée là aussi par M. Weiser dans la notion de "*Calm Computing*" [134]), c'est organiser la communication entre objets sans que l'utilisateur n'en ait conscience, et ni n'ait à intervenir. La transparence est la clé de l'immersion de l'utilisateur dans son environnement, ou, en inversant l'angle de vue, l'intégration automatique des "smart-objects" de l'utilisateur dans l'*intelligence ambiante*.

5.1.3 Les limites de l'abstraction matérielle

Jusqu'à présent, nous nous sommes intéressés à l'Internet des objets en tentant de résoudre la problématique de l'hétérogénéité de ses composants. L'abstraction matérielle que nous proposons libère des contraintes induites par les spécificités de chaque objet. Celui-ci peut alors être indifféremment remplacé par un autre offrant le même type de services. Cependant, ce qui caractérise le monde de l'Internet des objets, ce sont les échanges, les interactions entre les différents éléments [25] [69] [135]. L'abstraction du matériel et l'universalité de la programmation sont nécessaires pour parvenir à la création des briques logiques décrivant les actions de chacun des composants [39]. Mais elles ne sont pas suffisantes pour contrôler les échanges entre objets, ni même composer leurs interactions simplement. Or l'expérience globale d'un utilisateur de l'IoT est liée à ces interactions entre objets. La conformité de *SALT* aux besoins de description de la logique embarquée ne peut garantir la bonne correspondance entre messages émis et messages attendus. C'est l'organisation de l'imbrication des effets des comportements des objets mis en œuvre qui doit maintenant être résolue.

5.2 L'approche par l'abstraction des échanges

Nous proposons de nous intéresser à la complexité des échanges entre objets car ceux-ci, à notre sens, constituent le propre de l'IoT [69]. Les communications des objets *entre eux* façonnent une application tout autant que leurs comportements singuliers. Faire interagir des objets, c'est expliquer ce que chacun doit faire, mais cela en fonction de ce que font les autres.

5.2.1 L'abstraction des échanges

Parce que le nombre de types d'objets est considérable, les possibilités d'interactions sont infinies. Encore faut-il que ces interactions aient du sens. Ainsi, connecter un interrupteur et une ampoule est possible, mais dans quel objectif ? Nativement, on imagine que l'interrupteur puisse allumer et éteindre l'ampoule. Mais en fait, de très nombreuses réactions peuvent être déclinées à partir de la liaison entre ces objets. L'ampoule peut, par exemple, ne s'allumer qu'un instant, ou bien attendre une deuxième pression sur l'interrupteur. Dans un système plus complexe, c'est l'ampoule qui envoie une information à l'interrupteur, le prévenant qu'elle est allumée (à la suite d'un autre événement), afin qu'il puisse, en cas d'appui, commander un autre dispositif.

Le problème qui se pose alors est d'imaginer les imbrications de messages qui permettent de mener à bien ces interactions, ainsi que la mécanique du dialogue, et le scénario de l'échange [40]. C'est dans ce but que nous proposons d'abstraire les interactions entre objets. Abstraire, c'est normaliser et identifier en un *cas d'usage* (nommé pour des besoins de classification) un échange similaire, déjà recensé, et bien formalisé. Si les échanges deviennent génériques, alors les comportements déployés sur les objets pourront se référer à ces abstractions : ainsi l'objet réel sera caché par son comportement abstrait.

Par exemple, l'interaction avec un objet réagissant à l'échange "*allumer/éteindre*" pourra être déclenchée par tout autre capable d'"*allumer/éteindre*". Notre objectif est de créer un cadre normatif, et d'assurer un bornage des cas possibles, par la liste exhaustive des messages échangés et la définition exacte de leur interprétation. Ainsi simplifiés et catégorisés, les échanges entre objets peuvent être bien plus facilement instanciés, tout en offrant une large gamme de comportements possibles. Par exemple, un bouton peut proposer l'échange "*allumer/éteindre*" par une simple pression, mais aussi par une triple-pression, ou en dépendance d'une autre information. "*Allumer/éteindre*" peut aussi être invoqué par des codes exécutés sur des objets a priori non destinés à cet usage. On peut imaginer par exemple que cette abstraction, générée par l'accéléromètre d'un téléphone, allume, en cas de mouvement du téléphone, la diode d'un TelosB, et ce uniquement si un capteur de lumière (un second TelosB) l'a activé[1]. Le découplage entre le comportement de l'objet et ses interactions avec ses partenaires offre une grande liberté dans les réalisations, et la normalisation des échanges en fiabilise la composition.

La solution globale que nous proposons fait la distinction entre deux notions :

le **Comportement** (*Behaviour*) : décrit avec *SALT*, il correspond au traitement des événements par le FST interprété par *D-LITe* ;

les **Schémas d'Interactions** (*Interaction Patterns*) : c'est l'abstraction des échanges, c'est-à-dire la liste des messages compris/émis liés aux caractéristiques spécifiques de ce type d'interactivité.

1. Exemple de démonstration de la versatilité des compositions possibles, ces 3 objets interagissant pour créer une alarme antivol nocturne.

5.2.2 La typologie des échanges dans l'Internet des objets

Bien que les possibilités soient multiples, la majorité des échanges que nous avons recensés dans le cadre de l'Internet des objets reste assez concentrée autour d'un nombre limité de notions : *asservissements*, *activations*, *variations* (par exemple augmentation ou diminution d'une valeur), ou encore *échanges d'alertes* ou de *messages*. Alors que A. Barros et al. [28] ont classifié les échanges en termes de nombre d'acteurs impliqués et du contrôle de chacun, nous proposons une nouvelle classification concernant la sémantique et la syntaxe des échanges. La liste qui suit est volontairement réduite afin de ne pas multiplier les combinaisons possibles, tout en couvrant la majorité des besoins. Cette liste de schémas d'interactions est mise à profit dans BeC^3.

1. **Boolean Interaction** : c'est l'interaction la plus simple, elle consiste en une activation/désactivation. Ici, deux messages sont échangés, le premier (*on*) pour l'activation et le second (*off*) pour la désactivation.
2. **Bounded Counter** : ce *schéma d'interactions* exprime l'augmentation ou la diminution d'une valeur ou d'un niveau. Il permet de gérer curseurs, atténuateurs, et tout type de potentiomètres. Quatre messages reflètent ces variations : *up* et *down* altèrent la valeur courante tandis que *off* et *full* imposent la valeur minimum ou maximum. Ce *schéma* peut s'utiliser pour le contrôle de la luminosité, du volume sonore, ou encore du zoom d'une caméra.
3. **Coordinates** : son usage concerne tout ce qui a un rapport avec le pilotage multidimensionnel. Les changements de position s'expriment via les messages *north, south, west, east, up* et *down*. Un joystick, un pad ou une souris peuvent implémenter et utiliser ce *schéma d'interactions* en sortie, tandis qu'une caméra pourra, par exemple, l'imposer en entrée. [2]
4. **Toggle** : ce schéma agit comme une bascule (un drapeau) qui est soit levé soit baissé. L'unique message, *toggle*, indique un changement. Ce *schéma d'interactions*, bien que très basique et sémantiquement plus pauvre que *Boolean Interactions*, peut être utile pour des commandes simples comme des boutons d'alarme, des compteurs, etc.
5. **Send** : il permet de transmettre du contenu (passé en argument dans le message) aux autres objets. C'est aussi un moyen de s'adapter aux autres types d'applications orientées données (*data-centric*) telles que les plus classiques services web, qu'il s'agisse de micro-blogging par exemple, de réseaux sociaux ou de toute transmission de contenu texte ou multimédia. Ce *schéma d'interactions* ne fournit qu'un seul message (*send*) qui embarque le contenu voulu (texte, binaires, flux...). *Send("Full capacity reached")* est un exemple d'utilisation de ce schéma pour transmettre un texte.
6. **Notification** : il est utilisé pour prévenir les autres objets à propos d'un changement particulier de l'état de l'objet transmetteur (*notify(msg)*). Ce

2. Plus précisément, des appareils pourront exécuter des FST qui utilisent ces Interactions Patterns.

schéma peut sembler similaire à **Send**, quoiqu'il insiste sur la notion d'avertissement. Alors que **Send** ne transmet que du contenu (c'est donc ce qui prime dans ce type de messages), *notify(msg)* met l'accent sur la notification en elle-même, le message *msg*, optionnel, n'est là que pour documenter la raison de l'alerte. C'est le *changement d'état* ici qui attire l'attention, et sur lequel repose la sémantique de l'échange. Le schéma *Notification* sera utilisé pour émettre alarmes et avertissements comme par exemple dans *notify(Fire)*.

Le contenu de cette liste décrit les *Schémas d'Interactions* en usage dans les *comportements*, que ce soit en entrée, ou en sortie. Il devient alors possible de vérifier la cohérence d'une composition par la correspondance entre les *schémas d'interactions* attendus par le *comportement* présent sur un objet, et les *schémas d'interactions* fournis par les objets qui lui sont connectés. Si l'objet source "parle" le *schéma d'interactions* "compris" par l'objet cible, alors la composition a du sens[3]. Cette catégorisation des échanges entrant/sortant permet d'imaginer des usages très différents pour un même objet, qui pourra intervenir dans une large gamme d'applications.

Prenons l'exemple d'une caméra. Celle-ci, compatible D-LITe, est capable d'exécuter un *comportement* la pilotant. Ce *comportement* accepte en entrée les commandes du *Schéma d'interactions* "Coordinates". En installant D-LITe sur son ordinateur, un utilisateur peut la piloter avec sa souris grâce à un *comportement* fournissant "Coordinates" en sortie. La mise en relation de la souris et de la caméra va leur permettre d'interagir. Admettons qu'un problème rende la souris inutilisable, il suffira à l'utilisateur de choisir un autre objet disposant d'un *comportement* capable de générer "Coordinates". Imaginons qu'un tel comportement soit disponible pour son clavier. Après déploiement, en associant son clavier à la caméra, l'utilisateur pourra reprendre le contrôle. Il pourrait aussi bien utiliser son téléphone portable, pour peu qu'un *comportement* générant du "Coordinates" à partir de l'écran tactile existe, ou encore utiliser l'accéléromètre en tant qu'outil de contrôle de la caméra.

5.3 Les apports du Crowd Centric

La construction d'applications distribuées reste en général affaire de spécialistes. En SOA, c'est un architecte qui a la charge de bâtir l'application, en mêlant logique et utilisation de services web. Bien qu'il reprenne le concept de la SOA, BeC^3 propose une répartition des rôles à la fois plus simple et plus participative.

La figure 5.2 compare l'approche BeC^3 à la SOA traditionnelle. Habituellement, les services web sont complexes et fournis par des entités extérieures. L'*architecte* qui réalise l'application les assemble d'après la description des besoins des utilisateurs et suivant les fonctionnalités offertes par les services web. Cette méthode n'est ni agile, ni adaptée au cadre particulier de l'Internet des objets. Proximité de l'utilisateur, des objets et du comportement qu'il veut leur faire adopter, toutes ces particularités

3. Ce n'est pas l'objet en lui même qui "parle", mais bel et bien le *comportement* qui sera dynamiquement installé sur l'objet.

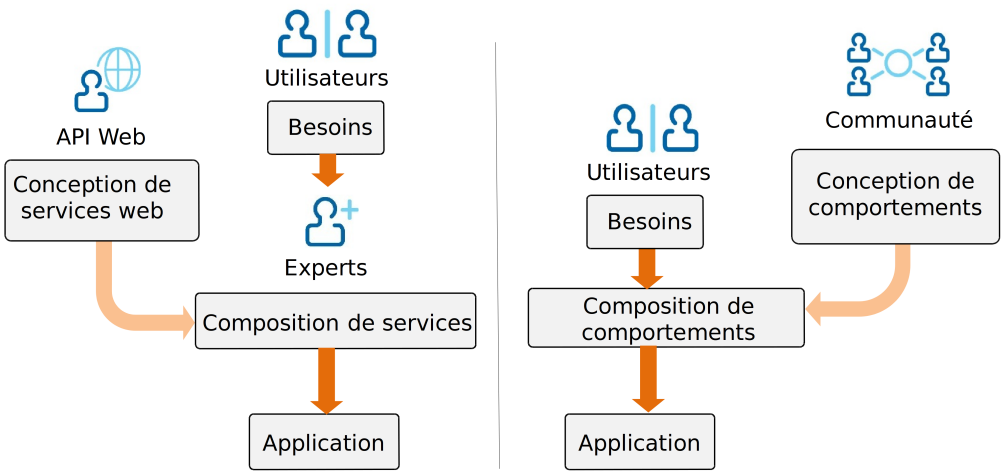

FIGURE 5.2 À gauche, la conception SOA classique dans laquelle un architecte combine les web-services créés par des informaticiens afin de réaliser une application conforme aux besoins exprimés par l'utilisateur. Dans BeC^3, l'utilisateur combine, lui même, directement, des éléments de logique fournis par une communauté d'utilisateurs.

amènent à repenser l'organisation. Nous plaidons pour un contrôle plus direct de l'utilisateur (nous sommes "*agiles*" en ce sens), et lui proposons de créer directement son application en la décrivant. Mais l'utilisateur n'a peut-être ni les compétences ni le temps nécessaires à la programmation intégrale de sa solution. C'est pourquoi BeC^3 est une solution de type "macro-programming" telle qu'inventoriée dans le panorama des méthodes de programmation pour les WSAN par R. Sugihara [123]. Ici, l'utilisateur combine les *comportements* proposés pour ses objets, et le moteur de cohérence de BeC^3 vérifie cette composition.

L'utilisateur juxtapose des petites unités logiques (des *comportements*) fournies par d'autres utilisateurs. C'est leur expertise qu'il utilise et qu'il organise selon ses besoins. BeC^3 est centré sur une communauté. Nous décrivons sa facette participative dans les paragraphes qui suivent.

5.3.1 Description du *Crowd centric*

Nous utilisons le terme *Crowd Centric* pour regrouper l'ensemble des structures participatives, en y mêlant le *crowd-sourcing* [77] (participation à la création) et le *crowd-founding* (financement participatif). Il s'agit du mode d'organisation, amplifié par l'Internet, dans lequel les contenus sont plus abondants, directement accessibles et plus interactifs. La distinction entre fournisseur et consommateur devient floue. Chacun peut intervenir, proposer, altérer la production partagée. Même si de nombreuses déclinaisons existent (plus ou moins dirigistes, plus ou moins sous le contrôle d'un système de validation, ou alors entièrement ouverte, proposant des contenus, des financements, etc.), ce modèle émergeant, favorisé par la diminution des "distances" (en terme d'accessibilité, de rôles, de hiérarchie) entre participants, connaît

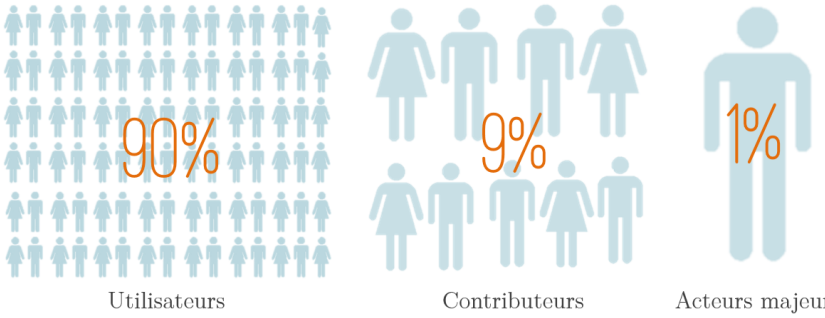

Figure 5.3 Répartition des utilisateurs de BeC^3. 90% se cantonnent à l'utilisation des ressources offertes par les 9% plus investis qui créent les *comportements*, tandis que le 1% restant réalise le portage de la machine virtuelle sur différentes plateformes.

des réussites notables. On pense par exemple à Wikipédia [4], le projet GNU [5] ou Linux [6] dans le domaine du savoir et de l'informatique, Folding@home et son projet de décryptage du génome humain de l'université de Stanford [7], ou encore à kickstarter [8] pour le financement de toutes sortes de projets (cinéma, industrie, startup, etc.). Si le réseau des réseaux n'est pas systématiquement à l'origine des processus participatifs, il facilite l'essor du *Crowd*, et certains auteurs évoquent un modèle d'affaires basé sur le web [77] ou d'intelligence collective [92].

Lorsqu'une certaine masse critique est atteinte, l'approche participative est un formidable accélérateur de l'offre, capable d'appréhender et de résoudre des problèmes complexes [36]. BeC^3 y recourt en proposant aux utilisateurs une liste de comportements qu'il met à leur disposition (figure 5.2). Les utilisateurs recensés ont la possibilité d'alimenter la base, afin d'offrir des *comportements* de plus en plus divers et riches.

5.3.2 Modèle participatif de BeC^3

Dans toute solution collaborative, l'investissement des membres est asymétrique [100], et ce d'autant plus que sa structure est informelle et peu hiérarchisée. Quelques participants sont acteurs et d'autres se contentent d'être plus passifs, profitant simplement du contenu fourni. Cette ligne de démarcation est intégrée dans la philosophie du système, sachant qu'à tout moment, chacun peut redéfinir son rôle au regard de ses envies, de ses compétences et de sa disponibilité. Cette répartition entre les différents usages (notamment sur l'Internet) a été étudiée par Brabham [36]. J. Nielsen a dénombré et décrit l'implication de chaque type de po-

4. http://www.wikipedia.org/
5. http://www.gnu.org/
6. http://www.kernel.org/
7. http://folding.stanford.edu/English/FAQ PS3
8. http://www.kickstarter.com/

pulation qui s'agglomère sur les organisations participative [100]. L'auteur en définit trois :

Les *visiteurs*. Ils consomment les ressources offertes sans en produire.

Les *contributeurs occasionnels*. Majoritairement consommateurs, il leur arrive de se consacrer au système en corrigeant ou améliorant les ressources.

Les *acteurs majeurs*. Créateurs des ressources, ils sont très investis.

L'auteur estime que le nombre des acteurs est inversement proportionnel à leur engagement. Aussi, selon lui, les *visiteurs* représentent 90% de la population globale, et ne génèrent qu'1% des apports au système. Les *contributeurs occassionnels* ne forment que 9% du total, mais fournissent à peu près le même pourcentage de ressources. Quant aux 1% restant (les *acteurs majeurs*), 90% des contenus leur sont dus.

En ce qui concerne BeC^3, nous répartissons (figure 5.3) les participants dans les catégories suivantes :

Les 1% sont les *développeurs/créateurs* des portages du framework *D-LITe* sur différentes plateformes. Très spécialisés, ils mettent en place l'abstraction matérielle sur les objets au moyen des langages de programmation supportés par le matériel visé. Leur niveau d'expertise accrédite leur capacité à proposer de nouvelles fonctionnalités dans le système, comme par exemple des extensions pour les *schémas d'interactions*.

Les *contributeurs occasionnels* (9%) partagent leurs FST qui enrichissent la bibliothèque de *comportements*. Ils utilisent le framework *D-LITe* et les *schémas d'interactions* qu'ils n'ont pas le droit d'altérer. Ils apportent leurs propositions de comportement d'objets, programmés en SALT (voir chapitre 4). Ils stipulent les contraintes en termes de *schémas d'interactions* recensés dans leurs *comportements*, puis partagent leur création.

Les *utilisateurs* se limitent à se servir de BeC^3. Ils n'ont pas besoin de connaissances en programmation, ils piochent dans les *comportements* disponibles pour leurs objets, et réalisent des interconnexions selon leurs besoins. Si la composition qu'ils décrivent est valide, elle est déployée sur les nœuds.

5.4 Les concepts mis en œuvre dans BeC^3

L'utilité du système tient, pour l'utilisateur, dans la facilité et la rapidité avec lesquelles il pourra lier les différents objets formant l'application et déployer le rôle de chacun d'eux. Pour y parvenir, nous avons dû caractériser les contraintes liées à la composition de services, et modéliser une solution capable à la fois d'exprimer les différents aspects des échanges entre objets tout en proposant une formalisation des contraintes à respecter pour obtenir une solution valide. Cette section décrit les composants de cette solution et les mécanismes de vérification utilisés.

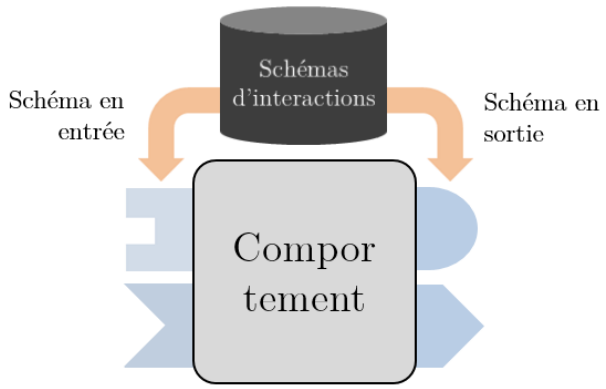

FIGURE 5.4 Un *comportement* dans BeC^3 est défini par son algorithme, mais aussi par la façon dont il interagit avec les autres. Les *schémas d'interactions* décrivent ces échanges typiques, et les entrées/sorties des *comportements* sont réduits à ces *schémas d'interactions*.

5.4.1 Éléments de BeC^3 : *features*, *comportements* et *schémas d'interactions*

Partant de l'expression des besoins, en l'occurrence offrir un système de composition d'algorithmes et de contrôle des interactions entre objets, BeC^3 se concentre sur ce que nous avons placé au centre de notre approche : *l'abstraction des échanges*. Les *schémas d'interactions* décrits dans la liste (5.2.2) caractérisent les principaux types d'échanges. Les *comportements* sont les algorithmes que les objets doivent adopter. Ces *comportements* dépendent des messages reçus et émettent eux-mêmes des messages afin d'interagir. Ils doivent préciser à quels *schémas d'interactions* ils réagissent, et ceux qu'ils génèrent. Le *comportement* présenté sur la figure 5.4 admet en entrée et émet en sortie un certain nombre de *schémas d'interactions*. Il faut donc vérifier que les attentes du *comportement* sont résolues, et que les messages émis sont bien pris en compte.

D'un autre coté, chaque objet impliqué dans l'application dispose lui-même de caractéristiques matérielles très précises. Grâce à D-LITe, celles-ci sont abstraites (voir l'abstraction matérielle du Chapitre 4). Nous les avons elles-mêmes classifiées en **features**. Un objet *D-LITeful*, quel qu'il soit, présente sa liste de **features** à BeC^3 lors de l'étape de découverte. Cette liste conditionne les *comportement* susceptibles d'être proposés pour cet objet.

Les *comportements* stipulent les *schémas d'interactions* requis et émis. La figure 5.5 dévoile la construction de l'ensemble avec deux objets sur lesquels sont exécutés des *comportements* adaptés à leurs caractéristiques (*features*). BeC^3 vérifiera ensuite la compatibilité de l'ensemble. Dans la figure 5.5, le *comportement* B impose le *schéma d'interactions* IP1, et accepte éventuellement IP3 (*strié*). IP1 est bien émis par le *comportement* A, et l'objet qui l'accueille fait bien partie des abonnés de l'objet 2. L'ensemble est cohérent, les attentes de chacun vérifiées, et l'application peut donc être lancée. La figure 5.6 insiste sur la concordance des

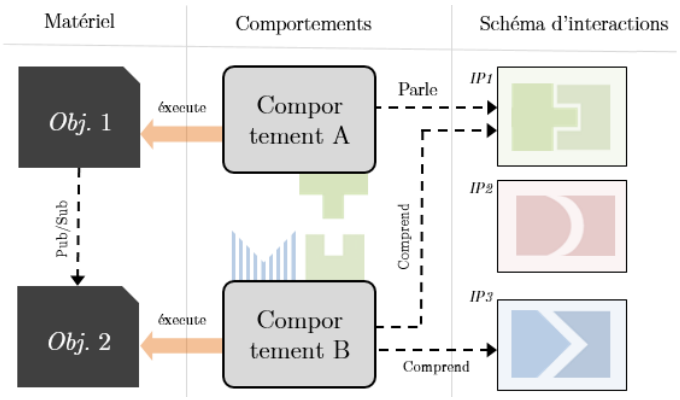

FIGURE 5.5 Deux Objects exécutant des *comportements* de BeC^3. Ils peuvent interagir car leurs *schémas d'interactions* coïncident. L'objet 2 exécute un *comportement* qui "comprend" IP1 (requis en entrée), et accepte optionnellement IP3. L'objet 1 lui exécute un *comportement* qui "parle" IP1.

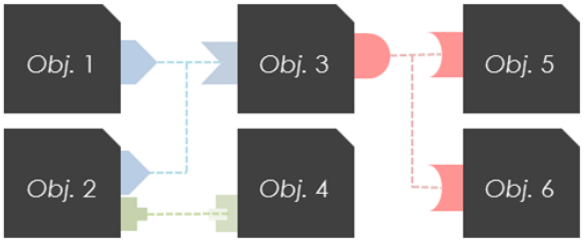

FIGURE 5.6 Une application BeC^3 met en jeu de multiples objets qui interagissent. Si les entrées/sorties correspondent, la chorégraphie est déployée et commence à s'exécuter.

échanges mettant en jeu un nombre plus important d'objets. Là encore, l'ensemble des contraintes est résolu, et l'application peut être déployée.

Sachant qu'un objet peut être en contact avec X autres, il faut trouver parmi ceux-ci de quoi résoudre ses besoins en termes de messages. Dans la description associée à chaque *comportement* dans le repository, on trouvera la liste des *schémas d'interactions* nécessaires ou optionnels en entrée et en sortie, ainsi que leur cardinalité. La cardinalité peut prendre les valeurs suivantes :

N impose une correspondance avec N *schémas d'interactions* dans les objets interlocuteurs ;

N + indique que le N est un minimum.

Grâce aux déclarations de la cardinalité des *schémas d'interactions* requis en entrée et en sortie par chaque *comportement*, des procédés de vérification vont pouvoir valider l'ensemble de la construction.

5.4.2 Modélisation et vérification

Lorsque l'utilisateur tente de déployer son application, il a, au préalable, défini les liens entre les objets, et indiqué les *comportements* choisis. Sachant que

chaque *comportement* définit ses contraintes en cardinalités d'entrée/sortie de *schémas d'interactions*, nous allons appliquer le processus de vérification qui suit pour valider l'ensemble.

Considérons la Chorégraphie C telle que $C = \{O, B, P\}$ dans laquelle :

O est l'ensemble d'objets impliqués ;

B le jeu de *comportements* déployés sur les objets ;

P les liens entre objets (relations d'abonnements).

Si nous nous intéressons à un élément C_x avec $0 < x < n + 1$ et $n = |O|$, nous avons : $C_x = \{o, \beta, A_x\}$, avec $o \in O$ l'objet exécutant le *comportement* $\beta \in B$ et $A_x (A_x \subseteq O^2)$ l'ensemble des relations de o avec les autres objets (couples d'objets, dont o est soit le premier membre, soit le second).

À partir de A_x, nous pouvons aussi établir la liste d'objets *précédents* O^- et la liste d'objets *suivants* O^+ de notre objet o. L'objet o est *précédé* par un objet o' si et seulement si $o' \in O^-$. Il est *suivi* par un objet o'' si et seulement si $o'' \in O^+$. Nous avons donc l'ensemble d'objets O^- qui alimente l'objet o, qui lui-même alimente l'ensemble O^+, ce que nous symboliserons par : $O^- \blacktriangleright o \blacktriangleright O^+$

Pour chaque *comportement* $\beta \in B$, les *schémas d'interactions* I s'organisent de la façon suivante :

I_{in}^{req} et I_{out}^{req} contiennent la liste des *schémas d'interactions* requis par β en entrée et en sortie respectivement. I_{in}^{req} et I_{out}^{req} sont des multi-ensembles, car un *comportement* requiert parfois plus d'une instance du même *schéma d'interactions* (voir la notion de *cardinalité*).

β a deux ensembles de *schémas d'interactions* optionnels I_{in}^{opt} et I_{out}^{opt} (repérés par le signe + dans leur cardinalité). Ici, la notion d'ensemble est suffisante (pas de redondance), c'est la présence d'un + qui est recensée.

Un *comportement* β, d'après les indications de son auteur, dispose donc de 2 ensembles et 2 multi-ensembles de *schémas d'interactions* :

$$I_{in}^{opt} = \{ip_0, ..., ip_n\}, I_{in}^{req} = \{\{ip_0, ..., ip_{n'}\}\}$$
$$I_{out}^{opt} = \{ip_0, ..., ip_{n''}\}, I_{out}^{req} = \{\{ip_0, ..., ip_{n'''}\}\}$$

Les contraintes de l'objet o obtenues, nous allons maintenant construire une description de ce qui est offert et requis par ses partenaires dans l'application.

Considérant que :

dans la Chorégraphie C, chaque tuple C_x indique que l'objet $o = O_x$ exécute le *comportement* $\beta = B_x$;

que β utilise les *schémas d'interactions* indiqués ;

qu'un objet o est précédé des objets de l'ensemble O^- et suivi par les objets de l'ensemble O^+ qui, respectivement, lui parlent ou l'écoutent.

Nous pouvons construire la liste OIP de *schémas d'interactions* sortant des objets O^+, déduite des deux éléments $OIP = \{OIP^{opt}, OIP^{req}\}$, avec :

$OIP^{opt} = \{ip_0, ..., ip_n\}$ son ensemble de *schémas d'interactions* optionnels ;

$OIP^{req} = \{\{ip_0, ..., ip_{n'}\}\}$ son multi-ensemble requis.

La construction de cet ensemble OIP^{opt} de *schémas d'interactions* optionnels sortants pour l'objet o obéit à la règle suivante :

$$OIP^{opt} = O_0(I_{in}^{opt}) \cup O_1(I_{in}^{opt}) \cup \ldots \cup O_n(I_{in}^{opt}) \text{ avec } O^+ = \{O_0, O_1, \ldots, O_n\}$$

(Toute occurence multiple d'un *schéma d'interactions* est supprimée par l'union des I_{in}^{opt}, puisque OIP^{opt} est un ensemble).

Du contenu des muti-ensembles de *schémas d'interactions* requis en entrée et en sortie des partenaires *précédents* et *suivants*, on détermine le nombre exact de *schémas d'interactions* émis en direction de notre objet o, et ceux requis par ses abonnés. Nous utilisons les sommes de multi-ensembles (et non l'union, car ici, le nombre d'instances de chaque *schéma d'interactions* est nécessaire) de la façon suivante :

$$OIP^{req} = O_0(I_{in}^{req}) \oplus O_1(I_{in}^{req}) \oplus \ldots \oplus O_n(I_{in}^{req})$$

OIP contient la liste des *schémas d'interactions* attendus par l'ensemble des objets suivants O^+. En suivant la même méthode, on fabrique la liste des *schémas d'interactions* entrants $IIP = \{IIP^{opt}, IIP^{req}\}$, bâtie à partir des ensembles de *schémas d'interactions* en sortie de tous les objets de O^-.

La construction de IIP et de OIP permet ensuite le lancement d'une vérification de l'application à déployer sur les appareils. En effet, en cas d'incohérence entre les *comportements* et les relations d'abonnements définies par l'utilisateur, nous pouvons considérer que l'application n'est pas fonctionnelle, puisque des échanges requis sont manquants.

La vérification s'obtient de la façon suivante : pour chaque objet o exécutant un *comportement* β, nous avons :

$$IIP \blacktriangleright o \blacktriangleright OIP \leftrightharpoons IIP \blacktriangleright \beta \blacktriangleright OIP$$

En remplaçant IIP et OIP par leurs contenus respectifs :

$$\{IIP^{opt}, IIP^{req}\} \blacktriangleright \beta \blacktriangleright \{OIP^{opt}, OIP^{req}\}$$

Puisque les entrées/sorties du *comportement* β sont décrites par les 2 ensembles et 2 multi-ensembles, nous obtenons les échanges suivants :

$$\{IIP^{opt}, IIP^{req}\} \blacktriangleright \{I_{in}^{opt}, I_{in}^{req}\}$$
$$\{I_{out}^{opt}, I_{out}^{req}\} \blacktriangleright \{OIP^{opt}, OIP^{req}\}$$

Grâce à ce modèle, l'outil de vérification et de déploiement de BeC^3 peut valider la Chorégraphie en utilisant les règles suivants :

1. $IIP^{req} \subseteq (IP_{in}^{opt} \oplus I_{in}^{req})$: Tous les *schémas d'interactions* requis dans O^- doivent être présents dans la liste des IPs compris par l'objet,
2. $I_{in}^{req} \subseteq (IIP^{opt} \oplus IIP^{req})$: Tous les *schémas d'interactions* requis par l'objet doivent être alimentés par O^-.

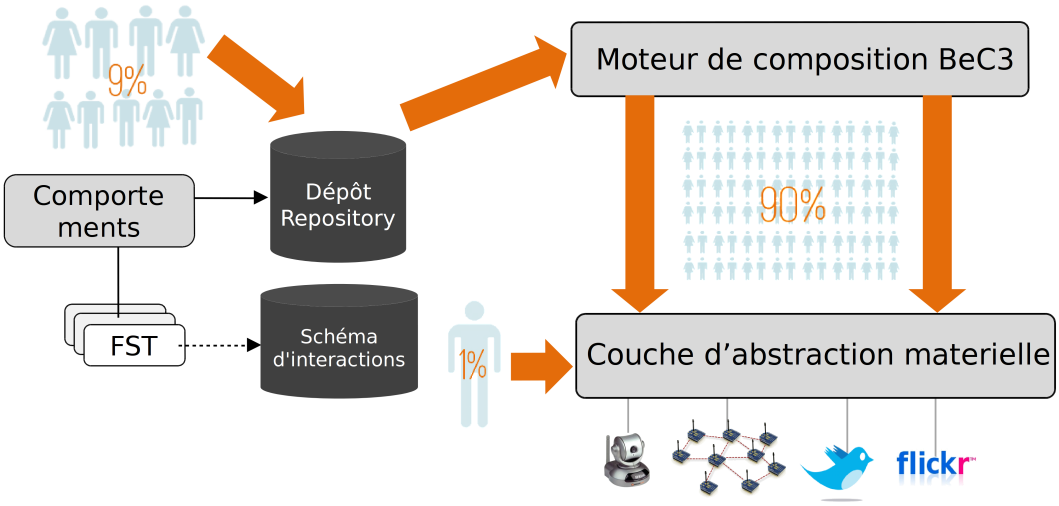

FIGURE 5.7 Dans la solution BeC^3, un utilisateur créé sans intermédiaire son application en composant les *comportements* proposés par une communauté de programmeurs (les contributeurs occasionnels, soit les 9%). Chacun des *comportements* est alors déployé et exécuté sur le nœud correspondant.

3. Le même raisonnement s'applique aux sorties de l'objet, qui doit vérifier que $OIP^{req} \subseteq (I^{opt}_{out} \oplus I^{req}_{out})$ et que $I^{req}_{out} \subseteq (OIP^{req} \oplus OIP^{opt})$.

Cette modélisation des échanges entre les différents éléments offre une méthode de validation des compositions réalisées par l'utilisateur, lui permettant de concevoir des réelles applications interactives. Même si le système ne garantit pas que toutes les applications créées auront du sens, au moins avons-nous vérifié la validité de l'ensemble des interactions et le respect des contraintes exprimées par les programmeurs.

Aussi BeC^3 permet-il à ces différents utilisateurs l'usage et l'alimentation du système selon leurs différents niveaux d'implication. La partie qui suit décrit l'architecture de la solution et sa prise en main par l'utilisateur.

5.5 L'architecture proposée et mise en œuvre pour la création d'applications

BeC^3 est une solution globale qui met en œuvre l'ensemble des travaux décrits jusqu'ici, à savoir D-LITe et SALT. La totalité des nœuds qui sont susceptibles de participer à l'application doit embarquer *D-LITe* qui est chargé de l'exécution des codes *SALT* requis par l'utilisateur. La figure 5.7 détaille l'organisation du système, ses différents constituants (*comportement*, *schémas d'interactions*), ainsi que les éléments accessibles et gérés par chacun selon son rôle (*utilisateurs*, *contributeurs occasionnels*, et *acteurs majeurs*).

Mais BeC^3 est avant tout conçu afin d'aider à la construction d'applications Internet des objets. Pour l'utilisateur, la partie visible de BeC^3 est un site web [9] sous forme de mashup, c'est-à-dire un outil d'aide à l'interconnexion d'éléments représentés graphiquement [68]. La page d'accueil du site invite l'utilisateur à s'identifier. BeC^3 récupère la liste des objets disponibles pour cet utilisateur. Une fois les objets intégrés à l'interface, l'utilisateur pioche pour chacun dans la liste des *comportements* disponibles pour composer son application, et tisse les liens entre objets. A sa demande, l'application est déployée sur les objets, qui se mettent à interagir.

Pour parvenir à cette apparente simplicité, l'ensemble se base sur une architecture d'éléments qui s'articulent autour des notions présentées précédemment, et ce afin de permettre la réalisation du mashup. Ce qui suit décrit l'organisation générale de l'architecture et les liens entre éléments.

5.5.1 L'architecture proposée

Les *comportements* (c'est-à-dire les FSTs) que les différents objets peuvent adopter sont stockés dans le dépôt (*repository*) (voir figure 5.7). Celui-ci est alimenté par les *contributeurs occasionnels* (les 9%) et les *acteurs majeurs* (les 1%). Les *comportements* qu'il contient sont destinés à être déployés sur les objets. Chacun d'eux est référencé par un nom et doté d'une description. La liste des *features* (voir la partie sur SALT 4.3 et la table 4.1) nécessaires à son exécution offre un système de pré-sélection : en effet, lorsqu'un utilisateur requiert l'usage d'un de ses objets, BeC^3 limite la liste des *comportements* proposés au vu des *features* dont il dispose. Aussi sa conformité au rôle à tenir est-elle vérifiée dès l'origine, lors de l'intégration sur l'espace de travail. L'utilisation de la notion de *feature* décorrèle à nouveau le code applicatif et l'objet. Un lien direct entre *comportement* et un modéle d'objet aurait été trop restrictif et peu "agile". Enfin, le créateur du *comportement* a transmis le code, exprimé au format XML du langage SALT décrit précédemment au paragraphe 4.3.4, et ses impératifs en terme de *schémas d'interactions*. L'ensemble de ces informations servira de base aux vérifications.

5.5.2 La mise en œuvre et l'interface avec l'utilisateur

L'interface avec l'utilisateur (les 90%) doit permettre, de manière simple, l'accès aux objets et à leur configuration. Celle proposée par BeC^3 est présentée sur la figure 5.8. Sur cette copie d'écran, un utilisateur est en train de créer une application avec 3 interrupteurs (boutons 1 à 3) et une lampe (Ampoule). Il a choisi de composer le comportement "*button toggle*" des boutons avec le comportement "*light toggle*" sur la lampe [10]. Les flèches indiquent les dépendances entre objets et les asservissements entre les différents comportements. Lors du déploiement, la cohérence de l'ensemble est contrôlée afin de vérifier que les contraintes de chaque *comportement*

9. Dont une instance est accessible à l'URL : `http://bec3.univ mlv.fr/`.

10. Ces objets sont vierges de tout comportement au démarrage. Les comportements choisis seront déployés lors du lancement de l'application.

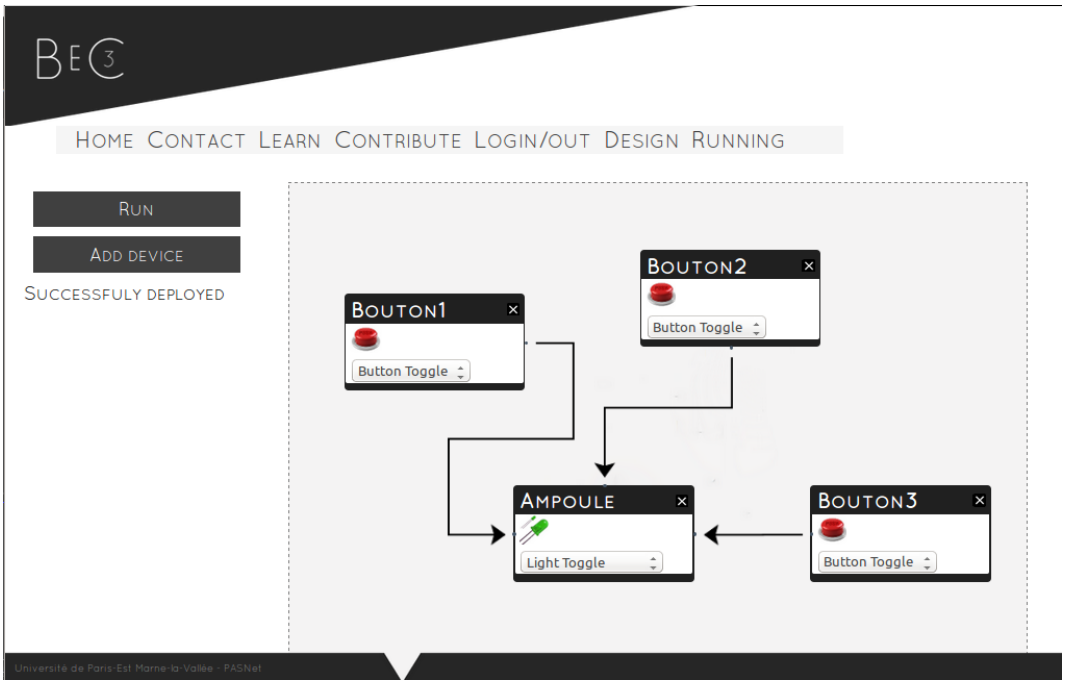

Figure 5.8 L'écran de conception de BeC^3 (le mashup) : on y voit l'ensemble des objets impliqués dans l'application, et les *comportements* choisis pour chacun d'eux. Les relations d'abonnement (*publish-subscribe*) sont symbolisées par des flèches. L'application (ici, un triple va-et-vient) est déployée en utilisant "*Send devices configuration*".

sont respectées. Si les assemblages sont validés, les codes SALT sont déployés sur les nœuds, et l'application démarre.

Si toutes les contraintes imposées par les *comportements* sont satisfaites, alors BeC^3 déploie l'ensemble sur les nœuds concernés. Dans le cas contraire, l'application n'est pas valide et ne sera pas déployée tant que les conflits ne seront pas résolus. La liste des dépendances à résoudre sera fournie à l'utilisateur. La capture d'écran de la figure 5.9 montre le résultat d'une vérification dans BeC^3. Par rapport à la structure qui fonctionnait précédemment sur la figure 5.8, deux erreurs ont été détectées : le bouton 1, sur lequel l'utilisateur veut déployer le *comportement* "Button On/off", a besoin d'un objet en écoute exécutant un *comportement* acceptant le *schéma d'interactions* "**Boolean Interaction**". La vérification échoue car cet IP n'est pas trouvé dans la liste des objets à l'écoute (Bouton2). Le bouton 2 n'a pas non plus le *schéma d'interactions "Toogle"* attendu en écoute (personne ne l'écoute), mais BeC^3 signale à l'utilisateur que l'ampoule l'implémente, et que les deux objets pourraient être connectés. Contrairement à l'application décrite en figure 5.8, le déploiement a été annulé et l'onglet "*Running*" est indisponible.

Afin que les différents utilisateurs investis puissent alimenter la base commune de leurs productions, le système comporte un back-office de gestion des *schémas d'interactions* et des *comportements*. La figure 5.10 montre l'aspect de cette partie accessible uniquement aux contributeurs occasionnels et aux acteurs majeurs. Cette

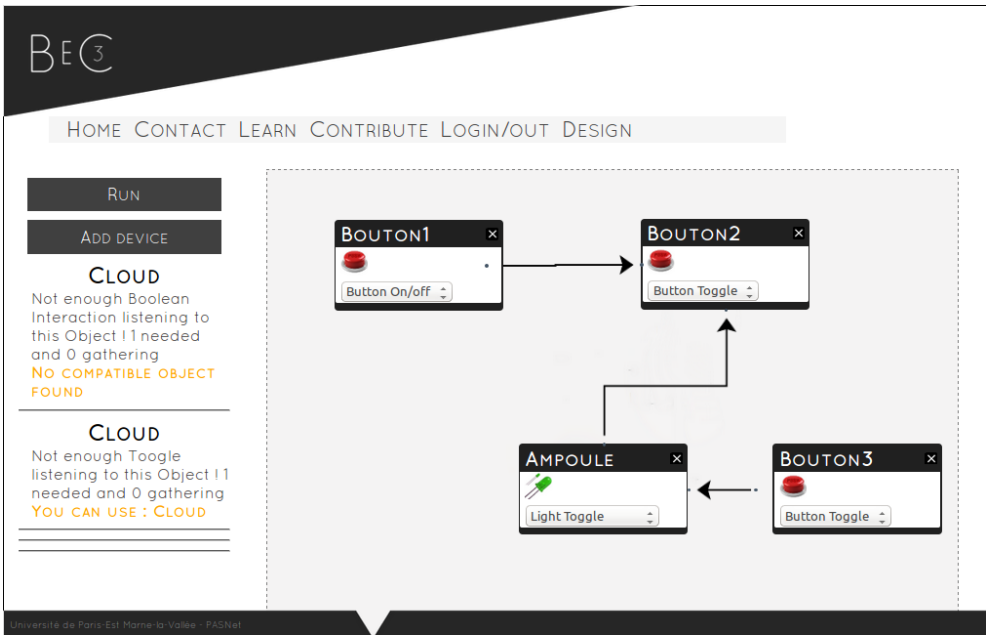

Figure 5.9 Vérification dans BeC^3 : Bouton1 a besoin d'un schéma d'interactions "*Boolean Interaction*". Bouton2 n'a pas d'objet adapté qui l'écoute. Le processus de vérification le signale, et propose d'utiliser l'ampoule.

partie est un accès à la base de données qui recense l'ensemble des éléments de BeC^3. C'est ici que les auteurs indiqueront les *features* indispensables pour les *comportements*, et détailleront les cardinalités des *schémas d'interactions* qu'ils requièrent et fournissent.

La partie fonctionnelle de BeC^3 contribue à la validation du système et à l'exploration de ses possibilités d'exploitation réelle et organisationnelle. L'aspect participatif autorise l'intégration des différents utilisateurs, et leur accorde la possibilité de tester le système et les liens avec leurs propres outils. Cela permet aussi de capitaliser les travaux des différents intervenants, et de récupérer les éléments déjà créés.

La partie qui suit présente la mise en œuvre de BeC^3 au travers d'un cas d'utilisation plus conséquent, dans le but de montrer la versatilité offerte par son approche de l'Internet des objets, et sa dynamicité adaptée à des besoins évolutifs.

5.6 L'illustration Smart-city de l'utilisation de BeC^3

Les villes intelligents (*smart city*) [76] fournissent de bons exemples de ce que les application de l'Internet des objets sont capables de réaliser. Ce type d'applications peut se construire grâce à des objets D-LITeful et avec l'assistance de BeC^3.

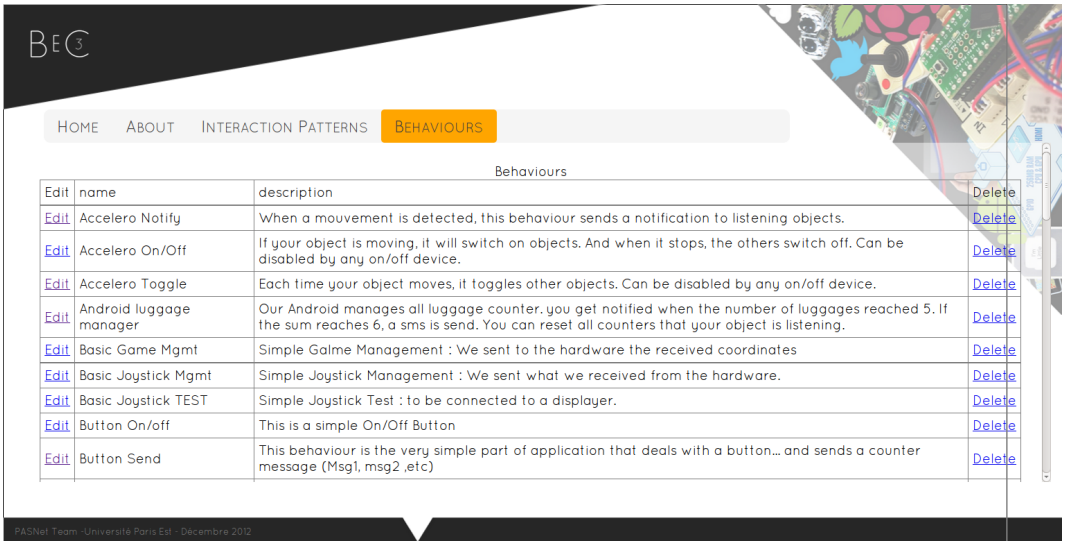

Figure 5.10 Dans cette partie, les différents utilisateurs référencés qui veulent s'impliquer peuvent ajouter, modifier ou supprimer des *comportements* (Behaviours) ou des *schémas d'interactions*.

Table 5.1 Un dépôt de *comportements* pour villes intelligentes

Nom	*IP* attendus	*IP* émis	description
Comportements pour **Capteur Place**			
PS1 : Std	-	Notif.	Place libre ou non
PS2 : countingFree	-	Bounded C.(1+)	Augmente ou diminue le nombre de places libres
Comportements pour **Panneau d'affichage**			
D1 : std	Bounded C.(1+).	-	Compte les informations reçues et affiche le total
D2 : std with alarm	Notif.(0+) Bounded C.(1+)	-	Comme D1, devient rouge en cas d'alarme
D3 : std, alarm and cascade	Notif.(0+) Bounded C.(1+)	Bounded C.(1+)	Comme D2, mais envoie le total compté en cascade
D4 : Global counter and send	Bounded C.(1+)	Send(1+)	Comme D2, mais envoie le total obtenu à un ou plusieurs autres objets
Comportements pour **objets Virtuels**			
Web1 : Push data to Web	Send(1+)	-	Met à jour le site web
Comportements pour **Détecteur incendie**			
F1 : Std	-	Notif.	Feu ou fumée détecté

5.6.1 Le scénario initial

Dans notre exemple, le gestionnaire du système d'information de la ville décide d'automatiser la publication des informations concernant les parkings. Chacun d'eux est équipé de panneaux d'affichage et chacune des places des parkings de la ville pourvue d'un capteur de masse métallique. Tous ces objets peuvent communiquer, puisqu'ils appartiennent au même projet Smart City, qu'ils disposent tous de notre framework *D-LITe*, et qu'ils sont bien configurés (au sens XMPP).

Les différents *comportements* disponibles et la description de chacun d'eux (rôle, *schéma d'interactions* mis en œuvre) sont fournis dans la Table 5.1. Au démarrage, chaque élément, bien que fonctionnel, ne fait rien de particulier, et n'interagit pas avec ses pairs. Le responsable du système d'information de la ville désire que chaque *détecteur de voiture* puisse envoyer des informations au panneau d'affichage. Afin d'obtenir le nombre de places libres, le *comportement* **PS2** (Table 5.1) doit être déployé sur les capteurs de place. Le *comportement* **D1** est quant à lui déposé sur chaque panneau d'affichage. Dans les parking, chaque afficheur traitera les notifications reçues des capteurs, et affichera le résultat obtenu. Afin de garantir la compatibilité des échanges, la sortie du *comportement* PS2 est conforme au *schéma d'interactions* **Bounded Counter**, ce qui correspond à l'entrée attendue par le *comportement* D1. Une fois ces *comportements* déployés, les habitants disposent, à l'entrée de chaque parking, d'informations sur son occupation.

5.6.2 L'évolution du scénario initial

Grâce à la flexibilité de notre architecture, une application plus évoluée peut facilement être déployée sans aucun accès physique aux appareils existants. Le gestionnaire du système peut ajouter des détecteurs de fumée compatibles D-LITe dans les parkings souterrains. Il assigne un nouveau rôle aux afficheurs : ceux-ci doivent continuer à afficher le nombre de places disponibles, mais aussi prendre en compte et afficher les alertes incendie en cas de détection de fumée. De nouveaux dispositifs sont ajoutés : un afficheur principal, au cœur de la ville, centralise les informations de l'ensemble des parkings. De plus, le site web de la ville doit être mis à jour en temps réel.

Dans ce scénario, les éléments sont organisés en cascades. Les capteurs de place envoient leurs messages à chaque unité d'affichage afin que celui-ci procéde à l'addition du nombre de places disponibles (en utilisant le *schéma d'interactions* **Bounded Counter**, comme précédemment). Et cette fois-ci, en choisissant le *comportement* **D3**, l'administrateur peut y connecter un système d'alarme qui relaie automatiquement les alertes vers le panneau. Le *comportement* de chacun d'eux est chargé de faire la cascade vers le panneau principal. Chaque évolution de la somme du panneau affecte ensuite l'afficheur principal qui, lui, exécute un *comportement* comprenant *Bounded Counter*. Ce dernier communique alors chaque modification au serveur Web de la ville grâce à l'objet virtuel adéquat. En résumé (voir figure 5.11), le gestionnaire du système d'information de la ville doit déployer :

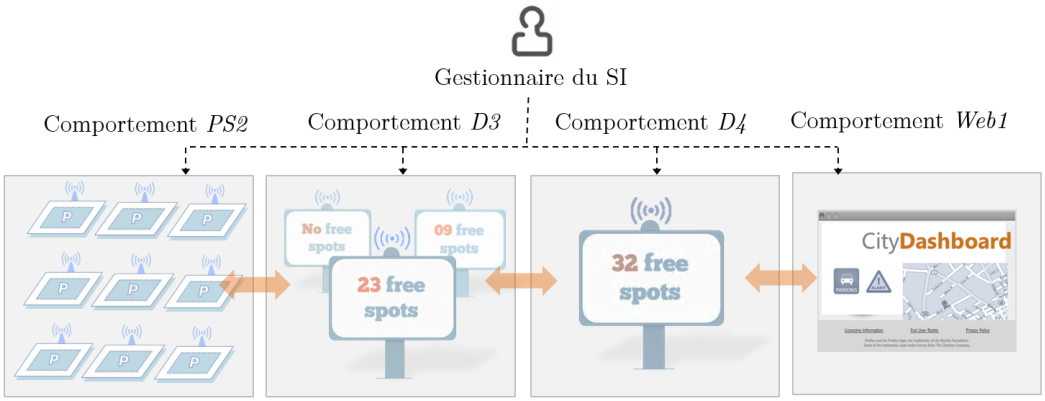

FIGURE 5.11 Dans une ville intelligente (*smart city*), le gestionnaire déploie les *comportements* choisis sur les objets correspondants dans le but de créer une application d'affichage des places de parking libres.

PS2 sur chaque capteur de place ;
F1 sur chaque détecteur de fumée ;
D3 sur les afficheurs, abonnés aux capteurs de voitures et aux détecteurs de fumée du parking dont ils ont la charge ;
D4 sur l'afficheur principal, abonné à tous les afficheurs de parking afin qu'il rende compte des évolutions du nombre de places disponibles ;
Web1 sur l'objet virtuel responsable du lien vers le site web de la ville. Cet objet dispose de toutes les autorisations nécessaires à la modification en temps réel du contenu des pages, ce qui permet un affichage dynamique du nombre de places disponibles en ville. Cet objet doit être abonné à l'afficheur principal, récupère les informations calculées par celui-ci et le met en page sur le site.

Chacun des algorithmes implémentés dans les *comportements* peut évoluer par l'ajout de nouvelles fonctionnalités et/ou l'emission ou la prise en compte de nouveaux messages en accord avec les *schémas d'interactions*. Le repository de BeC^3 permet à tous les contributeurs de proposer de nouveaux *comportements* ou d'améliorer ceux existants.

5.6.3 Les limites de Bec^3

BeC^3 utilise un vocabulaire volontairement restreint pour l'échange des messages. C'est une contrainte forte pour le concepteur de *comportements*. Il dispose d'une liste limitée de messages pour caractériser les entrées et les sorties de la logique qu'il doit décrire. Cette contrainte forte permet à BeC^3 l'exécution de vérifications plus efficaces de la compatibilité des *comportements* impliqués dans l'application.

Nous avons choisi de ne pas inclure l'identité de l'objet expéditeur dans les messages échangés, ce qui en allège le contenu. Nous pensons que cela offre à BeC^3 une grande généricité, et la capacité d'un passage à l'échelle. De la même façon, les cardinalités décrivant les attentes en *schémas d'interactions* permettent d'exprimer un

nombre strict ou une borne minimum, mais pas de maximum. Cet ajout compliquerait les algorithmes de vérification sans apporter, à notre sens, d'atouts significatifs en terme d'expressivité.

Mais ces choix peuvent conduire à des difficultés dans l'écriture des FST. Dans le cas, par exemple, de l'ajout de deux capteurs de contact chargés de savoir si les portes du parking sont ouvertes ou fermées, des limites peuvent apparaître. Si on désire savoir *combien de voitures ont été laissées dans le parking à sa fermeture*, nous pouvons connecter les capteurs de contact des portes à un compteur, et recenser le nombre de messages *la porte s'ouvre* et *la porte se ferme*. Si le compteur a obtenu deux messages *la porte se ferme*, alors le programmeur en déduit que toutes les portes sont fermées et il est possible de récupérer l'information du nombre de voitures toujours présentes, en émettant par exemple un message au panneau d'affichage général. Celui-ci, en cascade, mettra le site web à jour par exemple. Mais il est plus difficile de savoir quelle porte est fermée, car un message ne contient pas d'identification de son émetteur [11]. D'autre part, BeC^3 ne permet pas l'interrogation de l'état d'un objet, ou la demande de ré-emission d'une information (il utilise un mécanisme d'abonnement, et non de requête). BeC^3 a été conçu pour construire des chaînes de réactions automatiques (*réflexes*) dépendant des événements qui se produisent. Les mécanismes de requétage dynamique des objets correspondent plus à des applications WSAN [12].

En fait, BeC^3 n'est pas adapté aux approches données (*data-centric*). Parce qu'il est conçu pour l'Internet des objets, BeC^3 est *event-driven*, il s'intéresse principalement aux messages et événements [57]. C'est la sémantique de l'environnement et de ses caractéristiques qui est importante de notre point de vue. Dans BeC^3, l'accent est mis sur le sens des informations, et non leur valeur. Par exemple, nous désirons gérer des informations telles que *plein*, *chaud*, *vide*, *nouveau*, *plus*, *moins* ou encore le dépassement d'un seuil fixé, mais pas des valeurs scalaires réelles d'une mesure (*17*, *43*...), qui nécessitent, elles, une interprétation humaine. Toutefois, BeC^3 n'est pas intrusif. D'autres applications peuvent s'exécuter en parallèle, voire collaborer, tels DPWS [15], REST [62] ou d'autres [111] [95] [53], même propriétaires [13]. Il est tout à fait possible d'intégrer BeC^3 ou ses composants D-LITe et SALT au sein d'un autre système, et d'en faire usage ponctuellement, au gré des besoins.

5.7 Conclusion

La perception des applications de l'Internet des objets se limite souvent à des démonstrations de télécommandes domotiques. Pouvoir ouvrir une porte à distance, connaître la température d'une pièce puis demander l'allumage du chauffage à partir de son smartphone connecté à l'Internet est une solution intéressante, mais l'Internet des objets promet des applications plus complexes combinant de nombreuses

11. Des solutions existent cependant. Dans ce cas, l'évolution du *comportement* en lui ajoutant le *schéma d'interactions* "**Send**" indiquant le nom de l'émetteur résout l'ambiguïté.

12. Il serait cependant tout à fait possible d'agrémenter le GET des valeurs mesurées. Elles deviendraient alors accessibles par requête.

interactions et automatismes. La dimension *ubiquitaire* (omnipresente) et *pervasive* (*s'intégrant sans effort*) est un objectif majeur du domaine. L'intervention de l'être humain doit être réduite, limitée à la définition du service attendu, son rôle se cantonnant à l'expression de ses besoins a priori. Les objets doivent se charger des traitements qui en découlent. Dans notre exemple, c'est le capteur de température qui doit alerter le chauffage qu'un seuil a été atteint, et celui-ci prend la décision de s'allumer, l'utilisateur étant éventuellement le destinataire d'une notification lui rendant compte de ce qui s'est passé.

Dans ce chapitre, nous avons montré comment notre solution BeC^3 peut prendre en charge ce type d'applications. Les échanges entre objets, que nous avons appelé les *schémas d'interactions*, ont été caractérisés et recensés, selon les cas d'utilisation les plus courants. Cette classification conduit à normaliser le contenu de l'échange et à l'abstraire. Il devient alors possible de classer les *comportements* (le code à exécuter sur chaque objet) selon ses entrées/sorties en termes de *schémas d'interactions*. Enfin, en partant de la modélisation des contraintes fixées par le programmeur en termes de *schémas d'interactions*, nous avons défini les règles de vérification qui valident les compositions de *comportements* réalisées par l'utilisateur. L'application de ces règles garantit la bonne compréhension des objets entre eux, facilitant la création d'applications fonctionnelles, et évitant les incompatibilités.

Une plateforme complète est disponible afin d'expérimenter les concepts proposés (`http://bec3.univ-mlv.fr/`). Bâtie sur un mode participatif, en libre accès, elle permet aux utilisateurs non seulement de piocher dans la bibliothèque de *comportements* disponibles pour composer leurs applications, mais aussi d'alimenter cette bibliothèque, en fournissant de nouveaux codes ou en améliorant ceux qui sont fournis. La participation et l'implication des utilisateurs enrichissent le dispositif, et sa flexibilité permet au plus grand nombre de s'inclure, d'utiliser, ou d'interfacer ses propres applications avec notre solution.

Chapitre 6

Conclusion

> Il y a d'innombrables [...] choses qui sont offertes à disposition en permanence. C'est évidemment un aspect essentiel des choses d'être pour nous aussi continûment offertes à disposition.
>
> Dernières nouvelles des choses
> Roger-Pol Droit

Une étape est franchie lorsque des études quittent le cercle confidentiel des spécialistes d'un domaine pour atteindre un plus large public [1], laissant présager de l'engouement possible pour le sujet traité, ou, à tout le moins, de l'activité et de l'intérêt qu'il est susceptible de susciter. Cette thèse s'attache à résoudre des problèmatiques qui relèvent du domaine, en pleine expansion, de l'*Internet des objets*, à la croisée de nombreuses communautés scientifiques, dont les déclinaisons s'annoncent multiformes et les contours encore fluctuants.

6.1 Bilan

Les travaux présentés ici se concentrent sur une proposition fonctionnelle pour la construction d'applications IoT, basée sur un analyse verticale, commençant par l'étude des impacts sur les couches basses d'une organisation logicielle, pour aboutir à un niveau conceptuel fait d'abstractions (matérielle puis des échanges) masquant la complexité de la structure d'une application distribuée sur une structure hybride, constituée d'une multitude d'éléments dissemblables.

Jusqu'à présent, l'exploration des promesses de l'Internet de objets s'est concentrée sur la résolution des premiers obstacles limitant l'interconnexion entre les architectures et les protocoles habituels de nos réseaux, et les réseaux sans fil de capteurs et effecteurs qui se multiplient. L'interconnexion réalisée, c'est maintenant vers l'investigation de la physionomie propre au domaine qu'il faut s'orienter. La perception du domaine se cantonne trop souvent à des télécommandes évoluées, agissant sur un effecteur ou interrogeant un capteur via l'Internet, qui, pour spectaculaires qu'elles soient, ne modifient pas radicalement notre expérience du quotidien. L'âme de l'Internet des objets se situe peut-être plus dans l'essor d'interactions sans contrainte,

1. Articles sur l'*Internet des objets* : Août 2013 les *Échos*. 1ere quinzaine de septembre 2013 Les *Échos*, Le *Huffington Post* et le *Monde*.

transparentes, d'objets entre eux, à travers de multiples réseaux, plutôt que dans l'asservissement distant d'un appareil, et son contrôle direct par un utilisateur.

Dans cette thèse, nous nous sommes attachés à passer en revue des problématiques touchant aussi bien aux effets des applications sur les couches basses des WSAN qu'aux difficultés à composer des applications distribuées, en passant par le respect de l'hétérogénéité des éléments impliqués, tout en gardant toujours à l'esprit l'objectif d'organiser une réelle collaboration, fluide, entre objets, respectueuse à la fois des contraintes des infrastructures traversées comme des besoins des utilisateurs. Les travaux présentés ne prétendent pas être l'unique solution au problème, mais permettent l'élaboration d'une structure complète répondant aux interrogations de départ, soit la *construction universelle d'applications chorégraphiées pour l'Internet des objets*.

Dans le but d'élaborer une pile cohérente, nous nous sommes focalisés sur les problèmes suivants :

> Quel *mode architectural (orchestration ou chorégraphie) est le mieux à même d'être toléré* par les différents composants de l'Internet des objets ?
>
> Comment résoudre *les problèmes liés à la grande diversité des matériels et des protocoles* qui permettent de piloter les objets, afin d'offrir un socle commun, universel et simple, garantissant l'expressivité la plus large possible, au regard des besoins applicatifs de l'utilisateur ?
>
> Par quel dispositif introduire un cadre normatif et agile, garantissant au mieux *la cohérence des interactions entre objets*, au sein d'une infrastructure hétérogène et évolutive ?

Nos propositions de réponses à ces questions correspondent aux trois contributions présentées dans le manuscrit de cette thèse :

Architecture adaptée aux différents composants : l'Internet des objets est né de l'agrégation des réseaux de capteurs à ceux existants et validés par l'usage. Or, les WSAN sont soumis à d'importantes restrictions, notamment en termes de consommation d'énergie. Dans le chapitre 3, nous avons quantifié les apports des applications chorégraphiées, tels que pressentis dans la littérature, du point de vue des longueurs des chemins parcourus lors de la transmission de messages, et du nombre de nœuds requis par cette transmission, sachant que les émissions/réceptions sans fil dissipent une grande part de l'énergie. Celles-ci ont été comparées aux architectures orchestrées montrant le gain important des premières. Les architectures logicielles chorégraphiées, adaptées au domaine de l'Internet des objets, pour peu qu'on le considère sous l'angle d'une collaboration entre les divers éléments qui le composent, se révèlent utiliser des chemins entre nœuds d'un même réseau jusqu'à trois fois plus courts que leurs équivalentes orchestrées. Les expérimentations menées sur banc de tests confirment l'estimation du gain obtenu par l'analyse mathématique, compris entre 0 et 66% selon le profil de la structure arborescente du réseau.

Virtualisation de l'objet et standardisation de son accès distant : en introduisant une couche d'abstraction matérielle embarquée dans notre framework D-LITe, le chapitre 4 propose un couplage faible entre le matériel et les programmes à leur faire exécuter. Cette médiation ouvre la voie à une plus grande interopérabi-

lité des applications créées, la même logique pouvant être exécutée sur des matériels différents, mais offrant des fonctionnalités identiques. Cette logique s'exprime grâce à SALT, le langage que nous proposons, basé sur les transducteurs, pour leur simplicité à être abordés par le plus grand nombre, tout en couvrant les besoins de description des comportements attendus par les objets impliqués dans l'application de l'utilisateur. Les transducteurs, notamment par leurs alphabets d'entrée/sortie, construisent naturellement les chorégraphies évoquées dans la contribution précédente. Enfin, SALT introduit des prédicats dans les alphabets qui ne remettent pas en cause la nature même des transducteurs, mais qui dotent notre langage d'un supplément d'expressivité améliorant les compositions réalisées, la richesse des contrôles exprimables devenant alors proche de celle obtenue par l'intermédiaire d'applications orchestrées. L'accès distant au framework D-LITe des différents objets est normalisé par l'utilisation de REST, procurant alors une vision unifiée des différents éléments à l'œuvre. D-LITe devient alors la colonne vertébrale de l'ensemble, favorisant les chorégraphies d'objets dont la diversité est aplanie par une approche se concentrant sur les fonctionnalités offertes, et permettant le déploiement à volonté, via le réseau, d'une application distribuée en phase avec notre vision pervasive de l'Internet des objets. Une dernière partie s'attache à protéger les applications distribuées de leur faiblesse majeure sur un environnement peu fiable, à savoir la perte de cohérence. En proposant, à l'intérieur du framework, un dispositif de création de multiples procédures de re-synchronisation des objets, selon les besoins, et selon des structures indépendantes de surcouches d'arbres, D-LITe permet de maintenir l'application globale dans une marge d'erreur sous contrôle de l'utilisateur, après son arbitrage entre le niveau de précision qu'il désire obtenir et le surcoût engendré par les vérifications déclenchées.

Abstraction et modélisation des échanges : au travers d'une catégorisation des échanges entre objets impliqués dans des applications Internet des Objets, nous avons construit une architecture complète, intégrant notre solution D-LITe, capable d'abstraire les échanges entre éléments, et de fournir des méthodes garantissant la faisabilité de l'application ainsi construite par un utilisateur novice, au moyen d'un système de vérification des dépendances résolues (ou non) entre les algorithmes combinés dans la chorégraphie. Ré-utilisant l'apport de D-LITe qui universalise l'accès aux objets, et autorise l'expression de programmes liés principalement aux fonctionnalités génériques de chaque objet, BeC^3 donne la possibilité de stocker les codes qui décrivent ces *comportements* en leur associant diverses notions : pour chacun d'eux, l'auteur indiquera les *features* requises (c'est-à-dire les fonctionnalités génériques manipulées par ce code), et décrira les échanges abstraits qu'il met en œuvre (que nous avons appelés les *schémas d'interactions*, et dont nous fournissons la liste précise et stricte). BeC^3 permet non seulement de découvrir les objets d'un utilisateur, mais aussi de lui proposer les *comportements* recensés correspondant aux *features* de ses objets, et d'en assurer le déploiement (via D-LITe). L'utilisateur pourra décrire les dépendances entre ses différents objets, et BeC^3 se chargera de vérifier la cohérence de l'ensemble, au regard des schémas d'interactions déclarés pour chaque *comportement* par son concepteur. Si les contraintes exprimées sont

toutes résolues, c'est que l'application respecte, pour le moins, les interactions entre objets. BeC^3 autorise alors le déploiement du code SALT du *comportement* (donné par le concepteur) grâce aux fonctionnalités de configuration distante de D-LITe. L'approche collaborative de BeC^3 (dépôt des *comportements* par des contributeurs, qui seront ensuite intégrés par les utilisateurs dans leurs compositions) doit permettre, à terme, une grande richesse de l'offre, multipliant le nombre d'applications possibles, tout en contrôlant leur faisabilité.

BeC^3 est d'ores et déjà utilisable, et permet simplement de composer les *comportement* partagés, afin de voir fonctionner l'Internet des objets, et ce grâce à notre approche top-down, dans laquelle la conception guide la composition et l'installation des services voulus, et n'est plus limitée par la liste des services pré-existants. Les objets impliqués hébergent notre framework D-LITe afin d'être intégrés dans l'application, ce qui garantit un accès universel à la fois à leur configuration, mais aussi à leur mise en œuvre dans l'ensemble, par la capacité du framework à abstraire le matériel et à fournir un langage universel, basé sur les transducteurs. Au final, les applications créées prennent la forme de chorégraphies de services offerts par les objets, chorégraphies dont nous avons montré l'efficacité en ce qui concerne les économies d'énergie, lorsqu'elles sont à l'œuvre dans des réseaux fortement contraints de type WSAN.

6.2 Perspectives

Notre vision de l'Internet des objets, sous son angle événementiel, et de ses applications, combinaisons de machines virtuelles programmables à distance et organisées en chorégraphie de services, peut à son tour servir de cadre pour de futures extensions. Nous pensons notamment à ce qui avait été un des moteurs implicites du questionnement d'origine : tendre vers l'autonomique. La conception de systèmes capables de s'auto-configurer, s'auto-réguler, s'auto-protéger et s'auto-réparer pourrait bénéficier de l'universalité proposée par notre solution, de sa communication standardisée, et des capacités à s'abstraire des dépendances trop fortes au matériel, tout en y garantissant un accès le plus riche possible et le continuel enrichissement du comportement de chacun des composants par le déploiement de nouvelles stratégies sur chaque noeud. L'ensemble des solutions présentées ici peut aussi servir de socle pour la communication et la description de interactions entre éléments autonomiques.

D'autres pistes sont susceptibles d'être explorées : les besoins quant à la découverte des nœuds, le suivi de présence et les relations d'abonnements entre objets sont spécifiques à l'Internet des objets. Or, nous manquons de recul sur les effets et les points d'achoppement à ce niveau, tant en termes de fonctionnalités à offrir que de charge pour les différents services/serveurs à l'œuvre. L'engouement pour l'Internet des objets devrait permettre d'avoir plus d'informations sur les volumes mis en jeu, à la fois en termes de données, mais aussi d'interactions par l'intermédiaire des réseaux, et sur la charge liée à l'offre des services attendus. Le choix d'une démarche

participative pour la plateforme peut devenir un atout ici. Enfin, l'utilisation des protocoles actuels et l'avènement de nouveaux, s'attachant à résoudre les problèmes d'annuaire, de messages entre objets, de découverte, de suivi de présence, doit être évaluée et mise en concurrence avec de nouvelles idées.

Pour finir, l'ouverture de BeC^3 à différents utilisateurs permet de soumettre son évaluation à l'aune de différents besoins applicatifs, afin de découvrir les goulets d'étranglements entre ses multiples constituants, de définir de nouvelles problématiques et d'explorer les limites de la proposition, pour en faire naître de nouvelles approches, plus adaptées.

Bibliographie

[1] *C-implementation of coap*, http://libcoap.sourceforge.net/. 25, 74

[2] *Coapy, a python implementation of coap*, http://coapy.sourceforge.net/. 25, 74

[3] *Copper (cu), a firefox module to interact in coap*, https://addons.mozilla.org/fr/firefox/addon/copper-270430/. 25, 74

[4] *Digital living network alliance (dlna)*, http://www.dlna.org. 22

[5] *Forum upnp*, http://http://www.upnp.org/. 22

[6] *A java livrary implementing coap*, http://code.google.com/p/jcoap. 25, 74

[7] *Norme wsdl (web service description langage) du w3c*, http://www.w3.org/TR/wsdl. 17, 64

[8] *Oasis (organization for the advancement of structured information standards)*, http://www.oasis-open.org/. 16, 22

[9] *Soap version 1.2 (w3c)*, http://www.w3.org/2002/07/soap-translation/soap12-part0.html. 17, 22, 64, 65, 66

[10] *Soda (service oriented device & delivery architecture)*, http://http://www.soda-itea.org/. 22

[11] *Wsa (web services activity) du w3c*, http://www.w3.org/2002/ws/. 18

[12] *Zigbee alliance*, http://www.zigbee.org/. 13, 24, 44, 59

[13] *Zigbee profiles*, http://www.zigbee.org/Products/CertifiedProducts/ZigBeeHomeAutomation.aspx. 68, 120

[14] *Part15.4 : wireless medium access control (mac) and physical layer (phy) specifications for low-rate wireless personal aera networks (lr-wpans)*, IEEE Standard for Information Technology, 2003. 15, 24

[15] *Devices profile for web services (dpws)*, 2009. 22, 120

[16] *Fi-ware internet of things(iot) service enablement*, 2011. 29

[17] Kemal Akkaya and Mohamed Younis, *A survey on routing protocols for wireless sensor networks*, Ad hoc networks **3** (2005), no. 3, 325 349. 13, 16

[18] Ian F. Akyildiz, Tommaso Melodia, and Kaushik R. Chowdhury, *A survey on wireless multimedia sensor networks*, Comput. Netw. **51** (2007), no. 4, 921 960. 37, 39, 68

[19] I.F. Akyildiz and I.H. Kasimoglu, *Wireless sensor and actor networks : research challenges*, Ad hoc networks **2** (2004), no. 4, 351 367. 11, 15, 16, 36, 37, 38, 39, 44, 57, 59

[20] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, *A survey on sensor networks*, Communications Magazine, IEEE **40** (2002), no. 8, 102 114. 11, 13

[21] Ibrahim Al-Oqily and Ahmed Karmouch, *A decentralized self-organizing service composition for autonomic entities*, ACM Transactions on Autonomous and Adaptive Systems (TAAS) **6** (2011), no. 1, 7. 25

[22] B.Y. "Alkazemi and E.A." Felemban, *"towards a framework for engineering software development of sensor nodes in wireless sensor networks"*, "Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications", ACM, 2010, pp. 72 75. 40, 68

[23] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella, *Energy conservation in wireless sensor networks : A survey*, Ad Hoc Netw. **7** (2009), no. 3, 537 568. 14, 37, 52

[24] Jari Arkko, Danny McPherson, Hannes Tschofenig, and Dave Thaler, *Architectural considerations in smart object networking*, (2013). 31, 62, 72

[25] Luigi Atzori, Antonio Iera, and Giacomo Morabito, *The internet of things : A survey*, Computer Networks **54** (2010), no. 15, 2787 2805. 21, 29, 62, 100, 102

[26] Francisco J Ballesteros, Enrique Soriano, and Gorka Guardiola, *Octopus : An upperware based system for building personal pervasive environments*, Journal of Systems and Software **85** (2012), no. 7, 1637 1649. 29

[27] A. Barros, M. Dumas, and P. Oaks, *Standards for web service choreography and orchestration : Status and perspectives*, Business Process Management Workshops, Springer, 2006, pp. 61 74. 20, 40

[28] A. Barros, M. Dumas, and A. Ter Hofstede, *Service interaction patterns*, Business Process Management (2005), 302 318. 78, 104

[29] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al., *Manifesto for agile software development*, (2001). 100

[30] Abdelfattah Belghith, Wafa Akkari, and Jean Marie Bonnin, *Traffic aware power saving protocol in multi-hop mobile ad-hoc networks*, Journal of Networks **2** (2007), no. 4, 1 13. 14

[31] PJ Benghozi, S. Bureau, and F. Massit-Folléa, *L'internet des objets, quels enjeux pour les européens ?*, 2008. 21, 29, 36, 62

[32] Tim Berners-Lee, *Request for comment 1945 : Hypertext transfer protocol http/1.0*, 1996. 18

[33] Toni A Bishop and Ramesh K Karne, *A survey of middleware*, Computers and Their Applications, 2003, pp. 254 258. 62

[34] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard, *Web services architecture*, (2004). 18

[35] Athanassios Boulis, Chih-Chieh Han, Roy Shea, and Mani B Srivastava, *Sensorware : Programming sensor networks beyond code update and querying*, Pervasive and Mobile Computing **3** (2007), no. 4, 386 412. 29

[36] D.C. Brabham, *Crowdsourcing as a model for problem solving an introduction and cases*, Convergence : The International Journal of Research into New Media Technologies **14** (2008), no. 1, 75 90. 107

[37] A. Broring, T. Foerster, and S. Jirka, *Interaction patterns for bridging the gap between sensor networks and the Sensor Web*, Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, IEEE, 2010, pp. 732 737. 44, 65

[38] Niels Brouwers, Koen Langendoen, and Peter Corke, *Darjeeling, a feature-rich vm for the resource poor*, Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems, ACM, 2009, pp. 169 182. 27, 63

[39] Christian Buckl, Stephan Sommer, Andreas Scholz, Alois Knoll, Alfons Kemper, Jörg Heuer, and Anton Schmitt, *Services to the field : An approach for resource constrained sensor/actor networks*, Advanced Information Networking and Applications Workshops, 2009. WAINA'09. International Conference on, IEEE, 2009, pp. 476 481. 12, 16, 28, 44, 62, 66, 67, 83, 102

[40] T. Bultan, J. Su, and X. Fu, *Analyzing conversations of web services*, Internet Computing, IEEE **10** (2006), no. 1, 18 25. 73, 103

[41] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, and G. Zavattaro, *Choreography and orchestration : A synergic approach for system design*, Service-Oriented Computing-ICSOC 2005 (2005), 228 240. 20, 43

[42] F Cabral Pinto, P Chainho, N Pássaro, F Santiago, D Corujo, and D Gomes, *The business of things architecture*, Transactions on Emerging Telecommunications Technologies (2013). 30

[43] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi, *Automatic service composition and synthesis : the roman model*, IEEE Data Eng. Bull **31** (2008), no. 3, 18 22. 68

[44] A.P. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, and M. Zorzi, *Architecture and protocols for the Internet of Things : A case study*, Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, IEEE, 2010, pp. 678 683. 23, 26, 68

[45] Erdal Cayirci, *Wireless sensor and actuator network applications and challenges*, Autonomous Sensor Networks : Collective Sensing Strategies for Analytical Purposes (2013), 1 15. 11, 13, 16, 31

[46] Yen-Kuang Chen, *Challenges and opportunities of internet of things*, Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific, IEEE, 2012, pp. 383 388. 26

[47] Yunxia Chen and Qing Zhao, *On the lifetime of wireless sensor networks*, Communications Letters, IEEE **9** (2005), no. 11, 976 978. 15

[48] A. Chlipala, J. Hui, and G. Tolle, *Deluge : Data dissemination for network reprogramming at scale*, Class project, Berkeley, University of California, Fall (2003). 26, 73

[49] C.Y. Chong and S.P. Kumar, *Sensor networks : Evolution, opportunities, and challenges*, Proceedings of the IEEE **91** (2003), no. 8, 1247 1256. 11, 14

[50] Geoff Coulson, *What is reflective middleware*, IEEE Distributed Systems Online **2** (2001), no. 8, 165 169. 62

[51] F. Curbera, Y. Goland, J. Klein, F. Leymann, S. Weerawarana, et al., *Business process execution language for web services, version 1.1*, (2003). 20

[52] P.A.C. da Silva Neves and J.J.P.C. Rodrigues, *Internet Protocol over Wireless Sensor Networks, from Myth to Reality*, Journal of Communications **5**. 23

[53] John Domingue, Dieter Fensel, and Rafael González-Cabero, *Soa4all, enabling the soa revolution on a world wide scale*, Semantic Computing, 2008 IEEE International Conference on, IEEE, 2008, pp. 530 537. 28, 65, 120

[54] Falko Dressler, *A study of self-organization mechanisms in ad hoc and sensor networks*, Comput. Commun. **31** (2008), no. 13, 3018 3029. 13, 16

[55] A. Dunkels, B. Gronvall, and T. Voigt, *Contiki-a lightweight and flexible operating system for tiny networked sensors. local computer networks*, Annual IEEE Conference on, 0, 2004, pp. 455 462. 25, 26, 52, 75

[56] A. Dunkels, T. Voigt, and J. Alonso, *Making TCP/IP viable for wireless sensor networks*, Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session, Berlin, Germany, Citeseer, 2004. 24, 31

[57] S. Duquennoy, G. Grimaud, and J.J. Vandewalle, *The web of things : interconnecting devices with high usability and performance*, Embedded Software and Systems, 2009. ICESS'09. International Conference on, IEEE, 2009, pp. 323 330. 23, 25, 44, 120

[58] Simon Duquennoy, Gilles Grimaud, and J-J Vandewalle, *Smews : Smart and mobile embedded web server*, Complex, Intelligent and Software Intensive Systems, 2009. CISIS'09. International Conference on, IEEE, 2009, pp. 571 576. 32

[59] T. Erl, *Service-oriented architecture : concepts, technology, and design*, Prentice Hall PTR, 2005. 16, 64

[60] Dave Evans, *The internet of things*, How the Next Evolution of the Internet is Changing Everything, Whitepaper, Cisco Internet Business Solutions Group (IBSG) (2011). 1, 36

[61] R. Fielding, UC Irvine, et al., *Request for comment 2616 : Hypertext transfer protocol http/1.1*, 1999. 18

[62] Roy T. Fielding and Richard N. Taylor, *Principled design of the modern web architecture*, ACM Trans. Internet Technol. **2** (2002), 115 150. 19, 65, 66, 74, 120

[63] Vassilis Foteinos, Dimitris Kelaidonis, George Poulios, Vera Stavroulaki, Panagiotis Vlacheas, Panagiotis Demestichas, Raffaele Giaffreda, Abdur Rahim

Biswas, Stephane Menoret, Gerard Nguengang, et al., *A cognitive management framework for empowering the internet of things*, The Future Internet, Springer, 2013, pp. 187 199. 29, 30, 68

[64] Dominique Gaïti, *Autonomic networks*, vol. 5, Wiley. com, 2010. 30

[65] E. Gamma, R. Helm, R. Johnson, J. Vlissides, et al., *Design patterns*, vol. 1, Addison-Wesley Reading, MA, 2002. 73

[66] Carlos F García-Hernández, Pablo H Ibarguengoytia-Gonzalez, Joaquín García-Hernández, and Jesús A Pérez-Díaz, *Wireless sensor networks and applications : a survey*, IJCSNS International Journal of Computer Science and Network Security **7** (2007), no. 3, 264 273. 11, 14, 15, 62

[67] D.B. Green, R. Hulen, and J. Moody, *IPV6 sensor service oriented architecture*, Military Communications Conference, 2008, pp. 1 6. 27, 44, 62, 65

[68] D. Guinard and V. Trifa, *Towards the web of things : Web mashups for embedded devices*, Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain, Citeseer, 2009. 19, 65, 101, 114

[69] D. Guinard, V. Trifa, F. Mattern, and E. Wilde, *From the internet of things to the web of things : Resource-oriented architecture and best practices*, Architecting the Internet of Things (2011), 97 129. 32, 64, 74, 102

[70] Dominique Guinard, Vlad Trifa, Stamatis Karnouskos, Patrik Spiess, and Dominic Savio, *Interacting with the soa-based internet of things : Discovery, query, selection, and on-demand provisioning of web services*, Services Computing, IEEE Transactions on **3** (2010), no. 3, 223 235. 32

[71] Guinard, D. and Trifa, V. and Wilde, E., *A resource oriented architecture for the web of things*, Proceedings of IoT (2010). 21

[72] A. Hagedorn, D. Starobinski, and A. Trachtenberg, *Rateless deluge : Over-the-air programming of wireless sensor networks using random linear codes*, Proceedings of the 7th international conference on Information processing in sensor networks, IEEE Computer Society, 2008, pp. 457 466. 26, 73

[73] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava, *A dynamic operating system for sensor nodes*, Proceedings of the 3rd international conference on Mobile systems, applications, and services, ACM, 2005, pp. 163 176. 27

[74] Sonia Hashish and Ahmed Karmouch, *Deployment-based solution for prolonging network lifetime in sensor networks*, Wireless Sensor and Actor Networks II, Springer, 2008, pp. 85 96. 13

[75] K. Henricksen and R. Robinson, *A survey of middleware for sensor networks : state-of-the-art and future directions*, Proceedings of the international workshop on Middleware for sensor networks, ACM, 2006, pp. 60 65. 29, 62, 73

[76] José M Hernández-Muñoz, Jesús Bernat Vercher, Luis Muñoz, José A Galache, Mirko Presser, Luis A Hernández Gómez, and Jan Pettersson, *Smart cities at the*

forefront of the future internet, The future internet, Springer, 2011, pp. 447 462. 1, 116

[77] J. Howe, *The rise of crowdsourcing*, Wired magazine **14** (2006), no. 14, 1 5. 106, 107

[78] Li Jing, Zhu Huibiao, and Pu Geguang, *Conformance validation between choreography and orchestration*, Theoretical Aspects of Software Engineering, 2007. TASE'07. First Joint IEEE/IFIP Symposium on, IEEE, 2007, pp. 473 482. 43

[79] R. Kaplan and M. Kay, *Phonological rules and finite state transducers*, (1981). 67

[80] Lauri Karttunen and Kenneth R Beesley, *Twenty-five years of finite-state morphology*, Inquiries Into Words, a Festschrift for Kimmo Koskenniemi on his 60th Birthday (2005), 71 83. 67

[81] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, and C. Barreto, *Web services choreography description language version 1.0*, W3C Working Draft **17** (2004), 10 20041217. 20

[82] Matthias Kovatsch, Martin Lanter, and Simon Duquennoy, *Actinium : A restful runtime container for scriptable internet of things applications*, Internet of Things (IOT), 2012 3rd International Conference on the, IEEE, 2012, pp. 135 142. 32

[83] Matthias Kovatsch, Simon Mayer, and Benedikt Ostermaier, *Moving application logic from the firmware to the cloud : Towards the thin server architecture for the internet of things*, Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on, IEEE, 2012, pp. 751 756. 32

[84] M. Kuorilehto, M. Hännikäinen, and T.D. Hämäläinen, *A survey of application distribution in wireless sensor networks*, EURASIP Journal on Wireless Communications and Networking **2005** (2005), no. 5, 774 788. 62

[85] N Kushalnagar, G Montenegro, and C Schumacher, *Rfc 4919 : Ipv6 over low-power wireless personal area networks (6lowpans) : overview*, Assumptions, Problem Statement, and Goals (2007). 31, 45, 75

[86] Freddy Lécué, Yosu Gorronogoitia, Rafael Gonzalez, Mateusz Radzimski, and Matteo Villa, *Soa4all : an innovative integrated approach to services composition*, Web Services (ICWS), 2010 IEEE International Conference on, IEEE, 2010, pp. 58 67. 28

[87] Philip Levis and David Culler, *Maté : A tiny virtual machine for sensor networks*, ACM Sigplan Notices, vol. 37, ACM, 2002, pp. 85 95. 27

[88] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al., *Tinyos : An operating system for sensor networks*, Ambient intelligence, Springer, 2005, pp. 115 148. 26

[89] M. Liu, J. Cao, G. Chen, and X. Wang, *An energy-aware routing protocol in wireless sensor networks*, Sensors **9** (2009), no. 1, 445 462. 15

[90] Luca Mainetti, Luigi Patrono, and Antonio Vilei, *Evolution of wireless sensor networks towards the internet of things : A survey*, Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on, IEEE, 2011, pp. 1 6. 12, 23, 37, 62, 100

[91] Rafik Makhloufi, Guillaume Doyen, Grégory Bonnet, and Dominique Gaïti, *Impact of dynamics on situated and global aggregation schemes*, Managing the Dynamics of Networks and Services, Springer, 2011, pp. 148 159. 31

[92] Thomas W Malone, Robert Laubacher, and Chrysanthos Dellarocas, *The collective intelligence genome*, IEEE Engineering Management Review **38** (2010), no. 3, 38. 107

[93] Gerben G. Meyer, Kary Främling, and Jan Holmström, *Intelligent products : A survey*, Computers in Industry **60** (2009), no. 3, 137 148. 15, 21

[94] Mehryar Mohri, *Finite-state transducers in language and speech processing*, Computational linguistics **23** (1997), no. 2, 269 311. 66

[95] G. Moritz, E. Zeeb, S. Pruter, F. Golatowski, D. Timmermann, and R. Stoll, *Devices profile for web services in wireless sensor networks : adaptations and enhancements*, Emerging Technologies & Factory Automation, 2009. ETFA 2009. IEEE Conference on, IEEE, 2009, pp. 1 8. 62, 120

[96] L. Mottola and G.P. Picco, *Programming wireless sensor networks : Fundamental concepts and state of the art*, ACM Computing Surveys (2010). 15, 21, 39, 52, 68

[97] Gavin Mulligan and Denis Gracanin, *A comparison of soap and rest implementations of a service based interaction independence middleware framework*, Simulation Conference (WSC), Proceedings of the 2009 Winter, IEEE, 2009, pp. 1423 1432. 19

[98] W. Munawar, M.H. Alizai, O. Landsiedel, and K. Wehrle, *Dynamic TinyOS : Modular and Transparent Incremental Code-Updates for Sensor Networks*, Communications (ICC), 2010 IEEE International Conference on, IEEE, 2010, pp. 1 6. 26, 73

[99] A. Nayak and I. Stojmenović, *"wireless sensor and actuator networks : algorithms and protocols for scalable coordination and data communication"*, Wiley-Interscience, 2009. 16, 43, 57

[100] J. Nielsen, *Participation inequality : lurkers vs. contributors in internet communities*, Jakob Nielsen's Alertbox (2006). 107, 108

[101] Mike P Papazoglou and Willem-Jan Van Den Heuvel, *Service oriented architectures : approaches, technologies and research issues*, The VLDB journal **16** (2007), no. 3, 389 415. 17, 64

[102] C. Peltz, *Web services orchestration and choreography*, Computer (2003), 46 52. 20, 36, 40, 42

[103] A. Pintus, D. Carboni, A. Piras, and A. Giordano, *Connecting smart things through web services orchestrations*, Current Trends in Web Engineering (2010), 431 441. 25, 44, 62

[104] Mirko Presser, Payam M Barnaghi, Markus Eurich, and Claudia Villalonga, *The sensei project : integrating the physical world with the digital world of the network of the future*, Communications Magazine, IEEE **47** (2009), no. 4, 1 4. 28

[105] Nissanka B Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao, *Tiny web services : design and implementation of interoperable and evolvable sensor networks*, Proceedings of the 6th ACM conference on Embedded network sensor systems, ACM, 2008, pp. 253 266. 19, 66

[106] Daniele Puccinelli and Martin Haenggi, *Wireless sensor networks : applications and challenges of ubiquitous sensing*, Circuits and Systems Magazine, IEEE **5** (2005), no. 3, 19 31. 11, 13, 14, 43, 57

[107] Guy Pujolle, Hakima Chaouchi, and Dominique Gaïti, *Beyond tcp/ip : A context-aware architecture*, Network Control and Engineering for QoS, Security and Mobility, III, Springer, 2005, pp. 337 346. 23

[108] Z. Qiu, X. Zhao, C. Cai, and H. Yang, *Towards the theoretical foundation of choreography*, Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 973 982. 20

[109] Rayene Ben Rayana and Jean-Marie Bonnin, *Intelligent middle-ware architecture for mobile networks*, MobileWireless Middleware, Operating Systems, and Applications, Springer Berlin Heidelberg, 2009, pp. 43 57. 62

[110] Tobias Reusing, *Comparison of operating systems tinyos and contiki*, Sensor Nodes Operation, Network and Application (SN) **7** (2012). 26

[111] A. Rezgui and M. Eltoweissy, *Service-oriented sensor-actuator networks : Promises, challenges, and the road ahead*, Computer Communications **30** (2007), no. 13, 2627 2648. 28, 44, 62, 120

[112] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, AF Harris, and M. Zorzi, *SYNAPSE : A network reprogramming protocol for wireless sensor networks using fountain codes*, Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON'08. 5th Annual IEEE Communications Society Conference on, IEEE, 2008, pp. 188 196. 26, 73

[113] Nancy Samaan and Ahmed Karmouch, *Towards autonomic network management : an analysis of current and future research directions*, Communications Surveys & Tutorials, IEEE **11** (2009), no. 3, 22 36. 30

[114] Ahmad Sardouk, Leïla Merghem-Boulahia, and Dominique Gaiti, *Agent-cooperation based communication architecture for wireless sensor networks*, Wireless Days, 2008. WD'08. 1st IFIP, IEEE, 2008, pp. 1 5. 14

[115] Ahmad Sardouk, Rana Rahim-Amoud, Leïla Merghem-Boulahia, and Dominique Gaïti, *Power-aware agent-solution for information communication in wsn*, Telecommunication Systems **48** (2011), no. 3-4, 329 338. 16

[116] Kristian Selén, *Upnp security in internet gateway devices*, TKK T-110.5190 Seminar on Internetworking, 2006. 22

[117] Z. Shelby, *Embedded web services*, Wireless Communications, IEEE **17** (2010), no. 6, 52 57. 40, 44, 62

[118] Z. Shelby and C. Bormann, *6LoWPAN : The Wireless Embedded Internet*, Wiley, 2010. 24, 31, 53

[119] Z. Shelby, B. Frank, and D. Sturek, *Constrained application protocol (coap)*, An online version is available at http ://www.ietf.org/id/draft-ietf-core-coap-18.txt (2010). 25, 31, 74, 75

[120] Patrik Spiess, Stamatis Karnouskos, Dominique Guinard, Domnic Savio, Oliver Baecker, LMSD Souza, and Vlad Trifa, *Soa-based integration of the internet of things in enterprise services*, Web Services, 2009. ICWS 2009. IEEE International Conference on, IEEE, 2009, pp. 968 975. 32

[121] ITU Strategy and Policy Unit, *Itu internet reports 2005 : The internet of things*, Geneva : International Telecommunication Union (ITU) (2005). 1, 62

[122] J. Su, T. Bultan, X. Fu, and X. Zhao, *Towards a theory of web service choreographies*, Web Services and Formal Methods (2008), 1 16. 86

[123] R. Sugihara and R.K. Gupta, *Programming models for sensor networks : A survey*, ACM Transactions on Sensor Networks (TOSN) **4** (2008), no. 2, 1 29. 63, 106

[124] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé, *Vision and challenges for realising the internet of things*, Cluster of European Research Projects on the Internet of Things (CERP-IoT) (2010). 22, 25, 26, 29, 36, 40, 62, 68

[125] Zeldy Suryady, Usman Sarwar, and Mazlan Abbas, *A gateway solution for IPv6 wireless sensor networks*, 2009 International Conference on Ultra Modern Telecommunications Workshops, Ieee, 2009, pp. 1 6. 37

[126] Vlasios Tsiatsis, Alexander Gluhak, Tim Bauge, Frederic Montagut, Jesus Bernat, Martin Bauer, Claudia Villalonga, Payam M Barnaghi, and Srdjan Krco, *The sensei real world internet architecture.*, Future Internet Assembly, 2010, pp. 247 256. 28

[127] Nicolas Tsiftes, Joakim Eriksson, and Adam Dunkels, *Low-power wireless ipv6 routing with contikirpl*, Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, ACM, 2010, pp. 406 407. 13, 50, 53

[128] M.A. Uusitalo, *Global vision for the future wireless world from the wwrf*, Vehicular Technology Magazine, IEEE **1** (2006), no. 2, 4 8. 1

[129] M Hadi Valipour, Bavar Amirzafari, Kh Niki Maleki, and Negin Daneshpour, *A brief survey of software architecture concepts and service oriented architecture*, Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on, IEEE, 2009, pp. 34 38. 17, 18, 64

[130] Q. Wang, Y. Zhu, and L. Cheng, *Reprogramming wireless sensor networks : challenges and approaches*, Network, IEEE **20** (2006), no. 3, 48 55. 26, 73

[131] Xiaonan Wang and Huanyan Qian, *Research on all-ip communication between wireless sensor networks and ipv6 networks*, Computer Standards & Interfaces (2013). 31

[132] T. Watteyne, A. Molinaro, M. Richichi, and M. Dohler, *From manet to ietf roll standardization : A paradigm shift in wsn routing protocols*, Communications Surveys & Tutorials, IEEE (2010), no. 99, 1 20. 38

[133] Mark Weiser, *The computer for the 21st century*, Scientific american **265** (1991), no. 3, 94 104. 10, 36, 102

[134] Mark Weiser and JohnSeely Brown, *The coming age of calm technology*, Beyond Calculation, Springer New York, 1997, pp. 75 85 (English). 10, 102

[135] Erik Wilde, *Putting things to rest*, (2007). 28, 64, 102

[136] J. Yick, B. Mukherjee, and D. Ghosal, *Wireless sensor network survey*, Computer Networks **52** (2008), no. 12, 2292 2330. 11, 13, 14, 16, 37, 38, 59

[137] Mohamed Younis and Kemal Akkaya, *Strategies and techniques for node placement in wireless sensor networks : A survey*, Ad Hoc Networks **6** (2008), no. 4, 621 655. 13

[138] Jianliang Zheng and Myung J Lee, *A comprehensive performance study of ieee 802.15. 4*, 2004. 16

[139] Haiying Zhou, Danyan Luo, Yan Gao, and De-Cheng Zuo, *Modeling of node energy consumption for wireless sensor networks.*, Wireless Sensor Network **3** (2011), no. 1, 18 23. 15, 52

[140] Quan Zhou and Runtong Zhang, *A survey on all-ip wireless sensor network*, LISS 2012, Springer, 2013, pp. 751 756. 13, 31, 74

[141] Sébastien Ziegler, Cedric Crettaz, Latif Ladid, Srdjan Krco, Boris Pokric, Antonio F Skarmeta, Antonio Jara, Wolfgang Kastner, and Markus Jung, *Iot6 moving to an ipv6-based future iot*, The Future Internet, Springer, 2013, pp. 161 172. 24, 30

[142] M. zur Muehlen, J.V. Nickerson, and K.D. Swenson, *Developing web services choreography standards the case of rest vs. soap*, Decision Support Systems **40** (2005), no. 1, 9 29. 19, 20, 65, 74

Liste des publications

[1] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. D-lite : Distributed logic for internet of things services. In *IEEE International Conferences Internet of Things (iThings 2011)*, pages 16 24. IEEE, 2011.

[2] Sylvain Cherrier, Yacine Ghamri-Doudane, Stéphane Lohier, and Gilles Roussel. Services Collaboration in Wireless Sensor and Actuator Networks : Orchestration versus Choreography. In *17th IEEE Symposium on Computers and Communications (ISCC'12)*, page 8 pp, Cappadocia, Turquie, July 2012.

[3] Sylvain Cherrier, Yacine Ghamri-Doudane, Stéphane Lohier, and Gilles Roussel. SALT : a simple application logic description using transducers for internet of things. In *IEEE International Conference on Communications - Communication Software and Services Symposium (ICC'13 CSS)*, Budapest, Hungary, June 2013.

[4] Sylvain Cherrier, Ismail Salhi, YacineM. Ghamri-Doudane, Stéphane Lohier, and Philippe Valembois. Bec3 : Behaviour crowd centric composition for iot applications. *Mobile Networks and Applications*, pages 1 15, 2013.

[5] Abderrezak Rachedi, Stéphane Lohier, Sylvain Cherrier, and Ismail Salhi. Wireless network simulators relevance compared to a real testbed in outdoor and indoor environments. In *Proceedings of the 6th International Wireless Communications and Mobile Computing Conference*, IWCMC '10, pages 346 350, New York, NY, USA, 2010. ACM.

[6] Abderrezak Rachedi, Stéphane Lohier, Sylvain Cherrier, and Ismail Salhi. Wireless network simulators relevance compared to a real testbed in outdoor and indoor environments. *International Journal of Autonomous and Adaptive Communications Systems*, 5(1) :88 101, 2012.

Soumissions en cours

[1] Mihaela Brut, Ismail Salhi, David Excoffier, Sylvain Cherrier, Yacine Ghamri-Doudane, Nicolas Dumont, Mario Lopez Ramos, and Patrick Gatellier. When devices become collaborative. supporting device interoperability and behaviour reconfiguration across emergency management scenario. WF-IoT 2014 (Soumis).

[2] Sylvain Cherrier, Yacine Ghamri-Doudane, Stéphane Lohier, and Gilles Roussel. Fault-recovery and coherence in web of things choreographies. WF-IoT 2014 (Soumis).

Table des figures